Software (Systems) Architecture Foundations

Lecture #2

Architecture Styles

Alar Raabe

Recap of Last Lecture

Architecture with desirable properties doesn't emerge itself, it needs to be designed !

- Software Systems **Architecture** is a
  - *fundamental conception* of a software system in its
  - **environment** embodied in
  - **elements**, their **relationships** to each other and to the environment, and
  - **principles** guiding software system design and evolution
- Architecture is **important**
  - as a cause of certain properties → **designing architecture allows us to address concerns** and to achieve required and desirable properties of software systems
  - as fundamental conception of software system → **allows us to reason** (i.e. answer questions) about the software system and its properties, and foresee those properties without building and testing the actual system
- Software Systems **Architecture Description** is a
  - collection of **related** (corresponding) **models**, organized into cohesive groups of
  - synthetic (constructed) or projective (derived) **views**, defined by **viewpoints** according to the related set of **concerns** (defined in architecture framework)
- Architecture Description has **value** as
  - a document, it **provides guidance** for constructing and evolving the software system, and allows us to record and communicate our knowledge and decisions about the software system architecture
  - a model, it **allows us to reason** (i.e. answer questions) about the software system architectures (e.g. evaluate and compare architectures)

Content

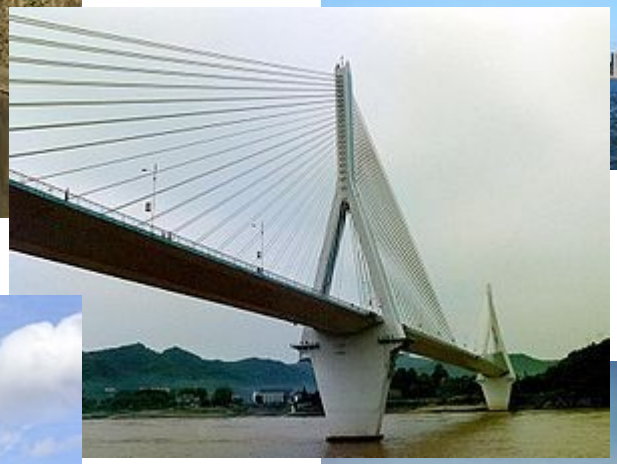
Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

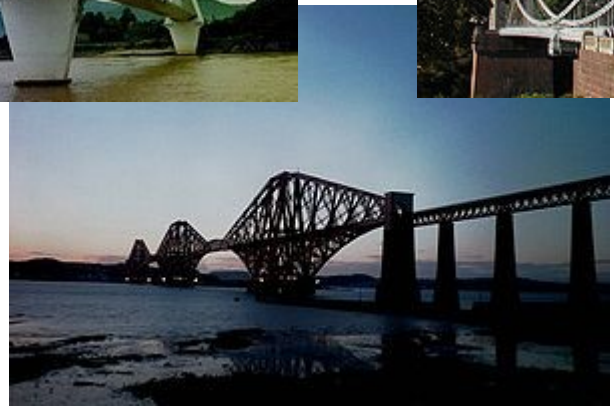
Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Example

Bridges … some have Something in Common

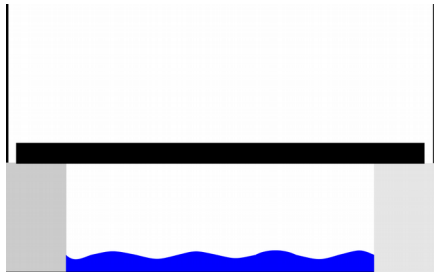
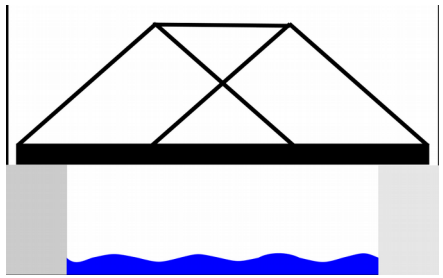
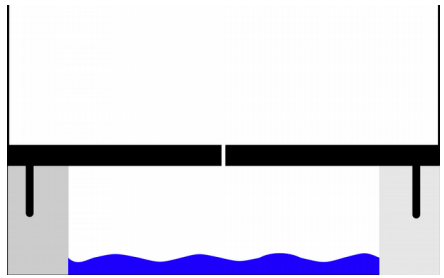




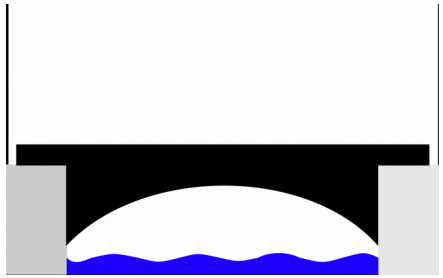
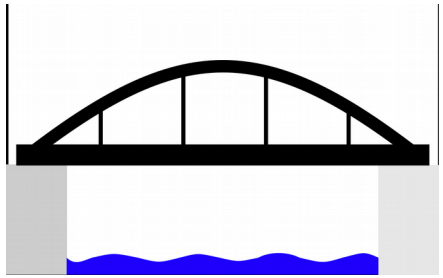
Example

Bridges Structure Types

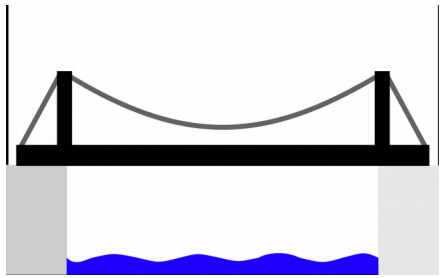
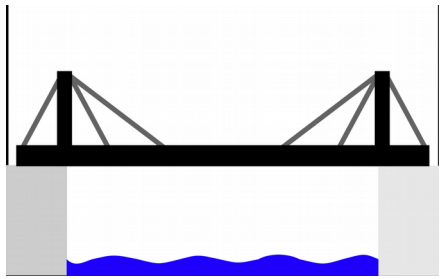
- Beam,
  Truss,
  Cantilever

- Arch,
  Tied arch

- Suspension,
  Cable-stayed

Different Levels of Commonality in Software

reuse of (design) knowledge

- Specific to a (programming) language
  - Software Idioms – coding/programming
    - describe usage of (programming) language for certain (simple) problems
  - Programming Style – programming
    - a consistent set of idioms (e.g. fluent style, functional style, …)
- (Programming) language independent
  - Design Patterns – design
    - describe standard solutions to certain common functional problems in certain contexts
  - Architecture Styles – architectural design
    - define a class of systems with specific (non-functional) properties
    - describe standard solution to a class of non-functional problems

Content

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

> Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction
>
> Roy Fielding

Software Architecture Style

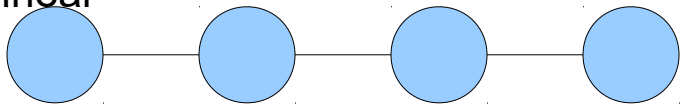
A coherent package of pre-made design decisions

- Characterizes **a family/class of system architectures** that are related by shared structural and semantic properties
- Software Architecture Style is defined by
  - a **vocabulary** of design elements
  - a set of **design rules**, or constraints (incl. **topology**)
  - semantic **interpretation** (incl. a computational model)
  - **analyses** that can be performed on systems built in that style
- Additionally Software Architecture Style is characterized by
  - the common examples of its use
  - the advantages and disadvantages of using it
  - the common specializations of it

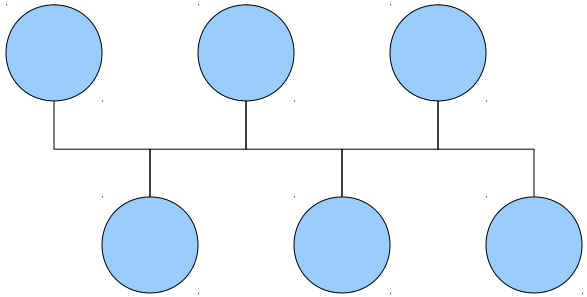
Various Topologies

What's the difference?

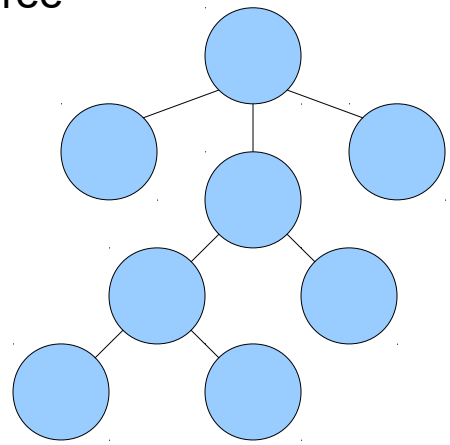
Linear



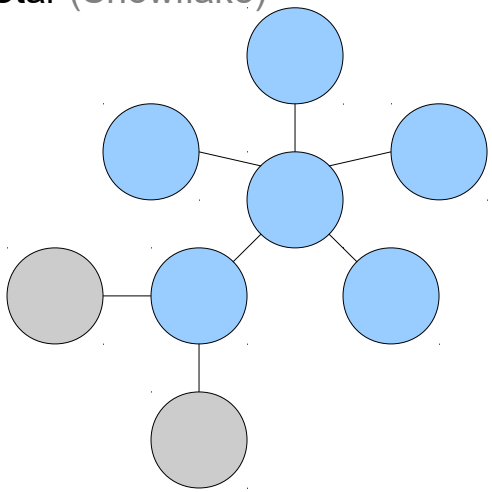
Bus



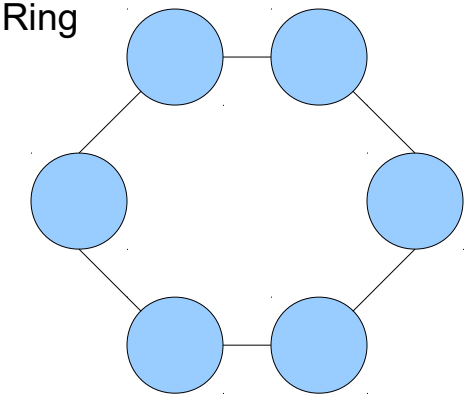
Tree



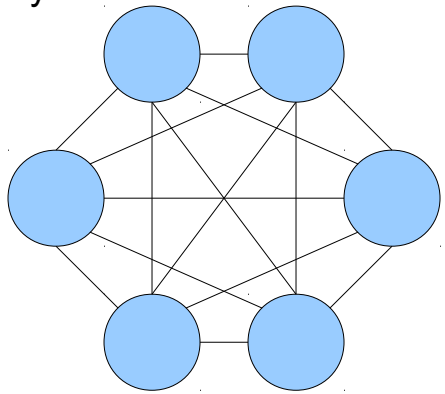
Star (Snowflake)



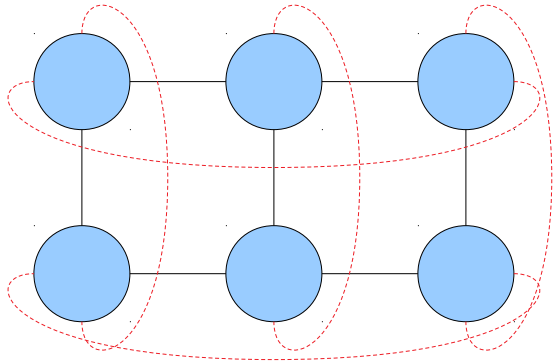
Ring



Fully Interconnected



Grid (Torus)



Benefits of using a Software Architectural Style

A coherent package of pre-made design decisions

- *Design Reuse*
  - well-understood solutions can be applied to new problems
- *Code Reuse*
  - shared implementations of invariant aspects of a style
- *Understandability of system organization (specific language)*
  - e.g. a phrase such as “client-server” conveys a lot of information
- *Interoperability*
  - supported by style standardization
- *Style-Specific Analysis*
  - enabled by the constrained design space
- *Visualizations*
  - style-specific descriptions matching engineer’s mental models

Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

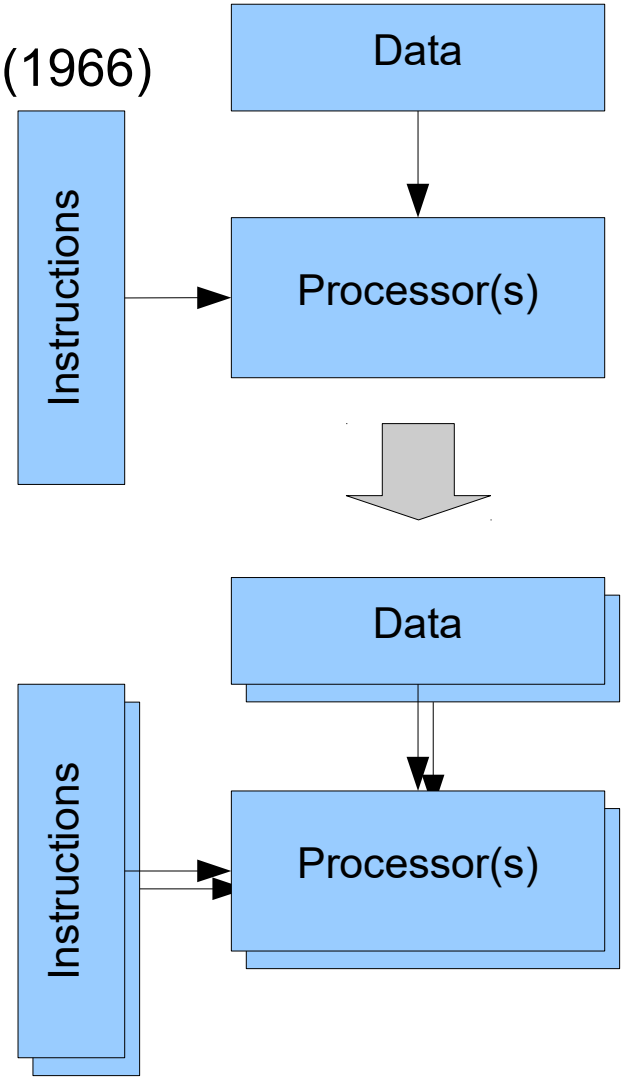
Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Flynn's Taxonomy of Computer Architecture

- Classification of computer architectures by M. J. Flynn (1966) based on two information flows
  - instructions stream
  - data stream

instructions / data	one	many
one	Single Instructions Single Data (**SISD**) traditional single processor	Multiple Instructions Single Data (**MISD**) processor pipelines & systolic arrays
many	Single Instruction Multiple Data (**SIMD**) vector and matrix processors	Multiple Instructions Multiple Data (**MIMD**) multiprocessors



Classifications of Software Architecture Styles

- "Boxology" (M. Shaw and P. Clements)
  - Major axes of classification are the **control and data interactions** among components
  - Architectural styles are discriminated by the following categories of features
    - which kinds of **components and connectors** are used in the style – constituent parts
    - how **control** is shared, allocated, and transferred among the components
    - how **data** is communicated through the system
    - **how data and control interact**
    - what type of **reasoning** is compatible with the style
- CMU SEI (L. Bass, P. Clements, R. Kazman)
  - Three kinds of architectural structures that embody decisions
    - how the system is to be structured as a set of code or data **units that have to be constructed or procured** (modules → units of implementation)
    - how the system is to be structured as a set of **elements that have run-time behavior** – (components → computation) and interactions (connectors → communication)
    - how the system will **relate to non-software structures in its environment**

Classification of Architecture Styles

"Boxology" – Shaw & Clements

- **Data Flow**
  - Topology is linear
    - Flow is continuous → Pipes and Filters
    - Flow is sporadic → Batch Sequential
  - Topology is arbitrary → Data Flow Network
- **Call and Return**
  - Topology is tree → Main Program & Subroutines or Layers (when data flow is isomorphic)
  - Topology is star → Client-Server
  - Topology is arbitrary → Abstract Data Types (static calls) or Objects (dynamic calls)
- **Independent Components**
  - Connectors are signals → Event-Based (asynchronous control)
  - Connectors are messages → Communicating Processes
- **Data Centered**
  - Connectors are queries → Repositories
  - Connectors are direct access → Black-Boards

Software Architecture Structures of Different Kind

CMU SEI

- **Module** structures – system's structure for construction or procurement
  - Decomposition structure → decomposition, information hiding and encapsulation
  - Uses structure → useful functional sub-sets (supports incremental development)
  - Layer structure → portability (supports change of the underlying computing platform)
  - Class (or generalization) structure → reuse and incremental addition of functionality
  - Data model → the static information structure
- **Component-and-connector** structures – system's structure during run-time
  - Service structure → independent development, modifiability
  - Concurrency structure → parallelism and issues associated with concurrent execution
- **Allocation** structures – relations to non-software structures in system environment
  - Deployment structure → performance, data integrity, security, and availability
  - Implementation structure → management of development activities and build processes
  - Work assignment structure → management of communications and the overall working process (also determines the major communication pathways among the teams)

Network-Based Software Architecture Styles

R. Fielding

- **Data Flow** Styles
  - Pipe and Filter (PF)
  - Uniform Pipe and Filter (UPF)
- **Replication** Styles
  - Replicated Repository (RR)
  - Cache ($)
- **Hierarchical** Styles
  - Client-Server (CS)
  - Layered-System (LS) and Layered-Client-Server (LCS)
  - Client-Stateless-Server (CSS)
  - Client-Cache-Stateless-Server (C$SS)
  - Layered-Client-Cache-Stateless-Server (LC$SS)
  - Remote Session (RS)
  - Remote Data Access (RDA)
- **Mobile Code** Styles
  - Virtual Machine (VM)
  - Remote Evaluation (REV)
  - Code-on-Demand (COD)
  - Layered-Code-on-Demand-Client-Cache-Stateless-Server (LCODC$SS)
  - Mobile Agent (MA)
- **Peer-to-Peer** Styles
  - Event-Based Integration (EBI)
  - C2 (Event-Based Layered Client-Server)
  - Distributed Objects
  - Brokered Distributed Objects

Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

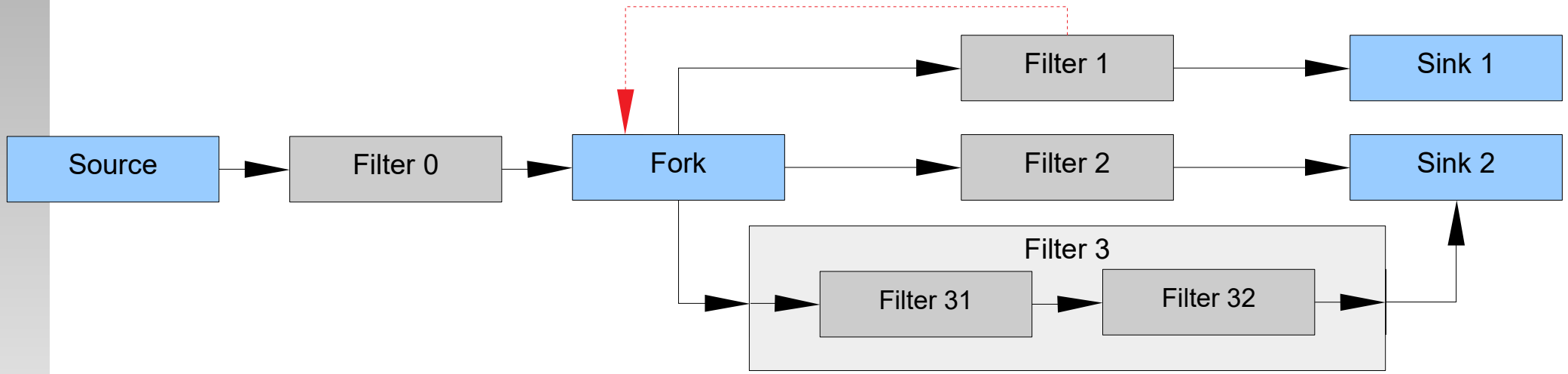
Analysis of some Software Architecture Styles

- For each style we look at
  - Structural pattern (components, connectors, *topology*)
  - Optional characteristics (constraints, invariants, computational model, theory, specializations)
  - Advantages and Disadvantages
  - Examples
- Main Architecture Styles
  - Data-Flow Systems – a.k.a. Pipes and Filters or Data Flow Networks
  - Data-Centered Systems – a.k.a. Repositories
  - Data Abstraction – e.g. Object-Oriented Systems
  - Event-Based – i.e. Implicit Invocation Systems
  - Independent Components – e.g. Service-Oriented Systems
  - Layered Systems – a.k.a. “Abstract Machines”
- Some “modern” Architecture Styles
  - Micro-Services, Map-Reduce, …
- Emergent Architecture – a.k.a “Big Ball of Mud”

Data-flow Systems

shared nothing!

- Structural pattern
  - Components – sources, filters, sinks, forks
  - Connectors – pipes
  - Topology – linear, tree, directed graph
- Characteristics
  - Constraints – is feedback (cycles) allowed or not, are pipes buffering or not, …
  - Invariants – filters are independent, and do not know the identity of other filters
  - Computational model – filters apply a local transformation to the input streams
  - Theory – Queueing Theory (A. K. Erlang 1909), theory of flow networks, …



Data-flow Systems – Evaluation

shared nothing!

- Advantages
  - Modifiability & Reuse – filters can be treated as black boxes (easy to add and/or replace)
  - Ease of construction – systems can be hierarchically composed (new filters can be created from existing)
  - Flexibility – system construction/configuration can often be delayed until run-time (supports late binding)
  - Scalability – filters can be run in parallel (they are isolated from other components of the system)
  - Understandability/Analyzability – system behavior is a simple composition of component behaviors (supports well certain analyzes like throughput, latency, deadlock)

- Disadvantages
  - Difficult to create interactive applications – because problem is decomposed into sequential steps
  - Common data representation – data has to be represented as the lowest common denominator
  - Parsing overhead (complexity) – every filter need introduce parsing/unparsing of the data stream
  - Unknown memory requirements and deadlock possibility – if output can't be produced before all input is received, filter will require a buffer of unlimited size (e.g. sort filter has this problem)
  - Often leads to a batch sequential organization of processing – because designer is forced to think of each filter as providing a complete transformation of input data to output data
  - Difficulties in data sharing and maintaining correspondences between separate, but related streams

Specializations of Data-flow Systems

"Boxology" – Shaw & Clements

- Pipe-and-Filter (pipeline) – data-flow networks restricted to linear topology
  - systems without feedback loops or cycles (acyclic)
  - pipelines (linear topology)
    - batch sequential systems – filters process all input data as a single entity
    - Unix pipes and filters – configured at run-time and pipes only handling ASCII streams
  - bounded pipes – restrict the amount of data that can reside on a pipe
  - typed pipes – require that the data passed between two filters have a well-defined type
  - systems with only fan-out components

- Data-flow networks – systems whose components operate on large, continuously available data stream
  - components are elements that asynchronously transform input into output with minimal retained state (i.e., transducers)
  - connectors are non-transforming high-volume data flow streams
  - components are organized in arbitrary topologies

Data-flow Systems Examples

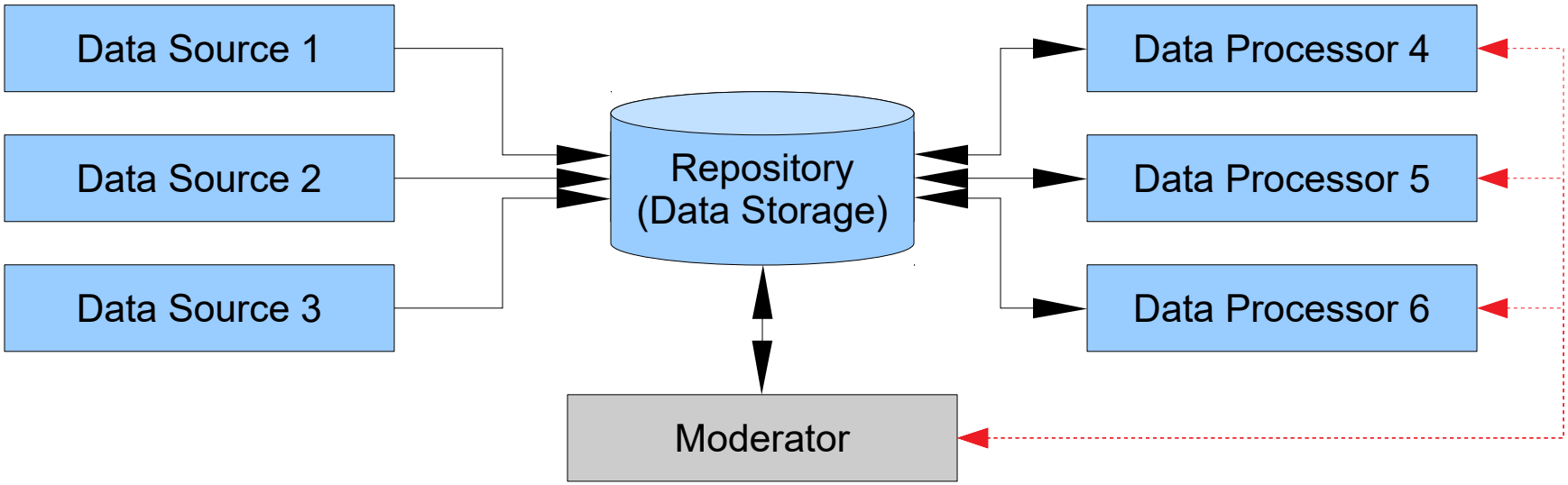
shared nothing!

- Pipes-and-Filters
  - Batch processing systems
  - Many traditional compilers (pipeline of lexical analysis, parsing, semantic analysis, and code generation)
  - Unix pipelines (defined by shell scripts)
- Signal and graphic processors
- Event Streaming (Complex Event Processing)
  - Apache Kafka
- Data Flow Networks (big data systems, stream processing)
  - Hadoop, Storm, Samza, Apache Flink, Reactive Streams
- Spreadsheets

Data-Centered Systems (Repositories)

shared everything!

- Structural pattern
  - Components – data sources/processors, repository (data storage), opt. moderator
  - Connectors – requests to and/or notifications from blackboard
  - Topology – star
- Characteristics
  - Constraints – transaction consistency
  - Theories – co-algebras, multi-stream interaction machines (Wegner), coordination theory, transaction theory, …



Data-Centered Systems (Repositories) – Evaluation

shared everything!

- Advantages
  - Scalability – easy to add more data sources and processors; data sources and processors can run in parallel and are synchronized through the central repository
  - Separation of concerns (problem partitioning) – each data source/processor performs separate function and solves part of the problem
  - Coupling – loose/low coupling between data sources and processors
  - Modifiability – data sources and processors can be modified independently
- Disadvantages
  - Coupling – tight coupling between data sources/processors and repository
  - Scalability – repository becomes bottleneck with too many data sources and processors
  - Difficult to analyze – non-deterministic behavior (system behavior emerges from the behaviors of data sources and processors)

Specializations of Data Centered Systems (Repositories)

Software Architecture
Shaw & Clements

- Traditional systems with central database – input stream of transactions is main trigger of the processes to execute

- Blackboard – current state of the central data structure is main trigger of selecting processes to execute
  - data sources = knowledge sources – separate, independent parcels of application-dependent knowledge (interaction among knowledge sources takes place solely through the blackboard)
  - repository = blackboard data structure – problem-solving state data, organized into an application-dependent hierarchy (knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem)
  - moderator = control – driven entirely by state of blackboard (knowledge sources respond opportunistically when changes in the blackboard make them applicable)

Data-Centered Systems (Repositories) Examples

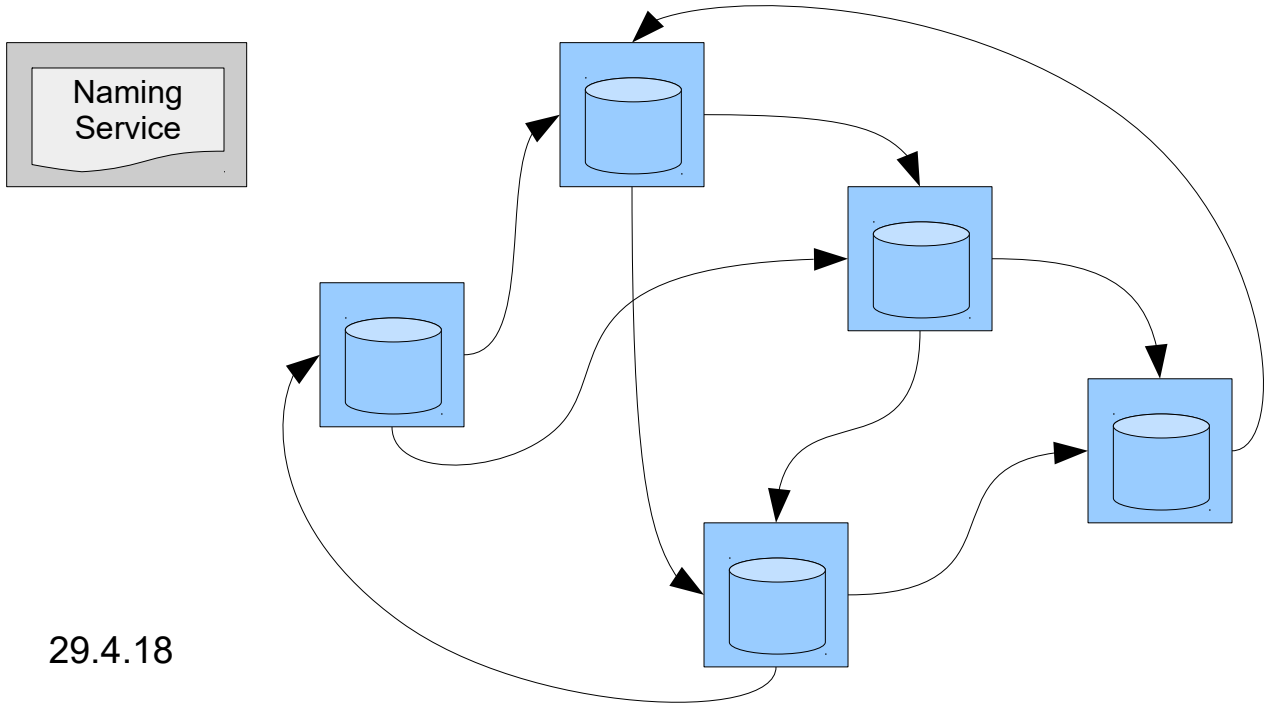
shared everything!

- Systems with global database
- Many language compilers & IDEs (with central AST or shared repository of programs and program fragments)
- Blackboard systems
  - Many expert systems (e.g. Hearsay II speech recognition system)
  - Applications for complex signal interpretations (e.g. speech and pattern recognition)
  - GBBopen (based on Common Lisp)
  - Blackboard Event Processor (based on Java)
- Shared (associative) memory (“tuple space”)
  - Java Spaces
- Systems that involve shared access to data with loosely coupled agents

Data Abstraction & Object-Oriented Organization

encapsulated data !

- Structural pattern
  - Components – objects (i.e. instances of the abstract data types), encapsulating data representations and their associated primitive operations, opt. naming service
  - Connectors – function and procedure invocations
  - Topology – arbitrary
- Characteristics
  - Constraints – object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), the representation is hidden from other objects, …



Object is a “manager” – it is responsible for preserving the integrity of a resource (a representation)

An inheritance relationship is not a connector, since it does not define the interaction between components in a system

Data Abstraction & Object-Oriented Organization – Evaluation

encapsulated data !

- Advantages
  - Modifiability – because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients
  - Separation of concerns (problem partitioning) – bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents
  - Understandability – supports direct modeling of object-oriented domain models (intuitive mapping from domain models)
  - Reuse – encourages software reuse
- Disadvantages
  - Coupling – in order for one object to interact with another (via procedure call) it must know the identity of that other object – whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it (supported by optional naming service)
  - Possible side-effects – if A uses B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa

Data Abstraction & Object-Oriented Organization

encapsulated data !

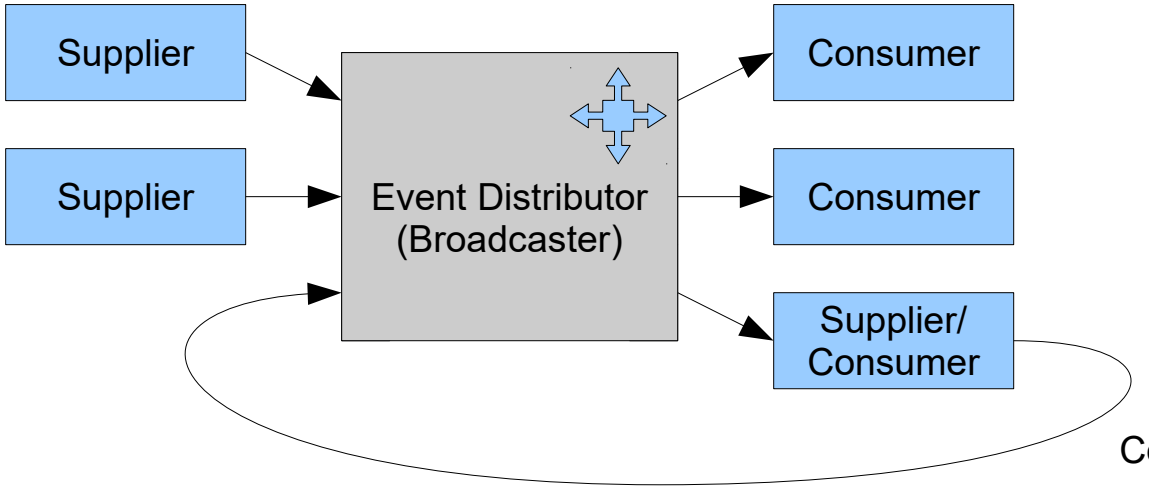
- Specializations
  - Objects can be concurrent tasks
  - Objects can have have multiple interfaces
  - Client-Server (client is triggering and server reactive)
  - Cooperating processes

- Examples
  - OO programming systems (Java, ...)
  - Distributed Object Systems (CORBA, SOM, COM/DCOM)
  - OSGi (Open Services Gateway initiative)

Event-Based, Implicit Invocation

decoupled control !

- Structural pattern
  - Components – event suppliers/generators, event consumers/listeners/handlers (can also generate events), event distributors/channels
  - Connectors – bindings between events and procedure invocations, procedure invocations (callbacks)
  - Topology – bus (star) or arbitrary
- Characteristics
  - Invariants
    - event suppliers do not know which consumers handle events – components cannot make assumptions about order of processing, or what processing will occur as a result of events they generate
  - Computational Model – event/driven “implicit” invocation of procedures



Event-Based, Implicit Invocation – Evaluation

decoupled control !

- Advantages
  - Coupling – loose coupling
  - Reuse and Flexibility – any component can be introduced into a system simply by registering it for the events of that system
  - Modifiability – components may be replaced by other components without affecting the interfaces of other components in the system (eases system evolution)

- Disadvantages
  - Non-deterministic – components don't have control over the computation performed by the system – when a component announces an event, it has no idea what other components will respond to it (even if it does know what other components are interested in the events it announces, it cannot rely on the order in which they are invoked, nor can it know when they are finished)
  - Performance – sometimes data can be passed with the event, but in other situations event systems must rely on a shared repository for interaction (in these cases global performance and resource management can become a serious issue)
  - Understandability/Analyzability – reasoning about correctness can be problematic, since the meaning of a procedure that announces events depend on the context of bindings in which it is invoked

Event-Based, Implicit Invocation

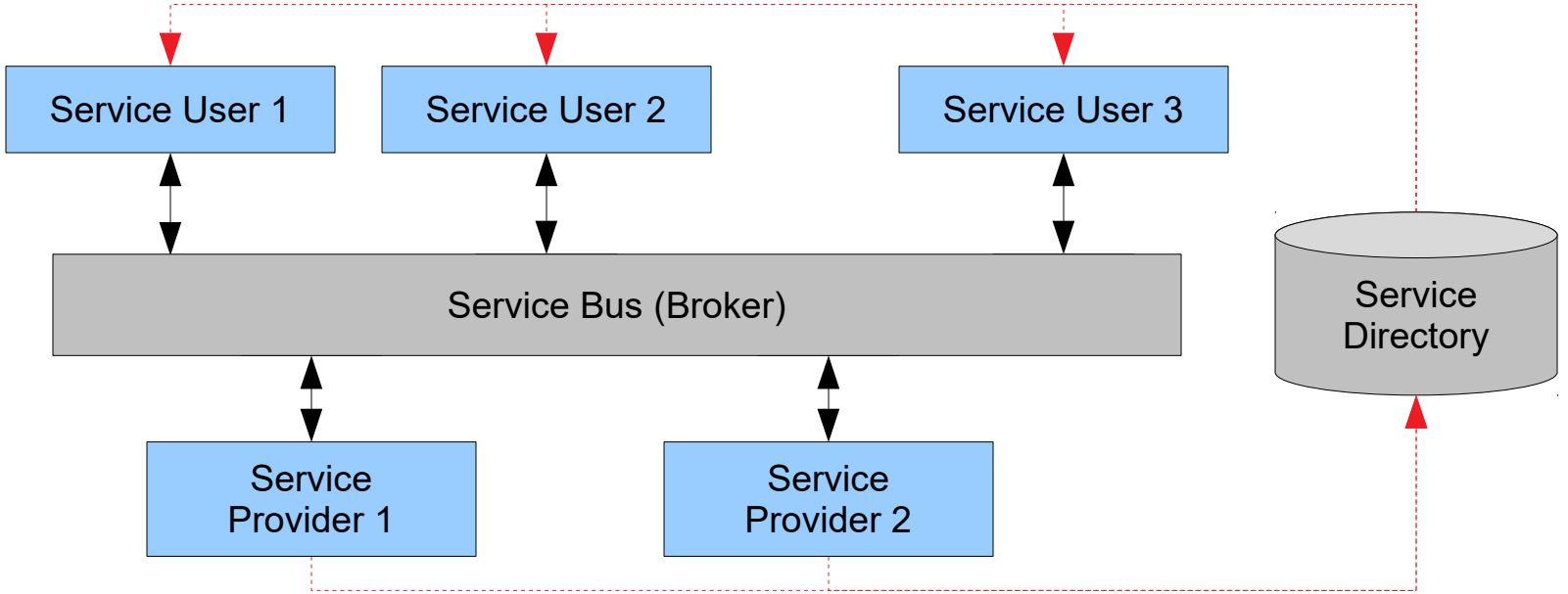
decoupled control !

- Specializations
  - Model-View-Controller (MVC) – specific types of event suppliers/consumers
  - Complex Event Processing
- Examples
  - User interfaces to separate presentation of data from applications that manage the data (GUIs, windowing systems)
    - X Window System
  - Programming environments (IDEs) to integrate tools and to support incremental semantic checking
  - Database management systems to ensure consistency constraints
  - Exception handling systems
  - Android Intent sub-system

Independent Components (e.g. Service Oriented Architecture)

bus and directory are optional !

- Structural pattern
  - Components – service providers, services users/consumers, opt. bus/broker, opt. directory
  - Connectors – synchronous and asynchronous calls, messages
  - Topology – bus (star)
- Characteristics
  - Constraints – (explicit) remote calls
  - Theories – CSP (C.A.R. Hoare), π-calculus (Milner, Parrow), …



Independent Components – Evaluation

bus and directory are optional !

- Advantages
  - Coupling – loose coupling (especially, if asynchronous calls are used)
  - Interoperability – service users can transparently call services implemented in disparate platforms using different languages
  - Modifiability – loose coupling between service users and service providers (services are self-contained and modular)
  - Extensibility – if bus is used, adding new services is easy
  - Reliability – good fault tolerance, if asynchronous calls are used
- Disadvantages
  - Performance – network overhead, overhead of intermediaries (like service directory), message parsing overhead
  - Scalability – limited scalability if synchronous calls are used
  - Security – difficult to achieve end-to-end security (needs message level security mechanisms)
  - Testability – difficult to test (complex)
  - Reliability – complex error recovery might be needed

Independent Components Examples

bus and directory are optional !

Killed by the Firewall !

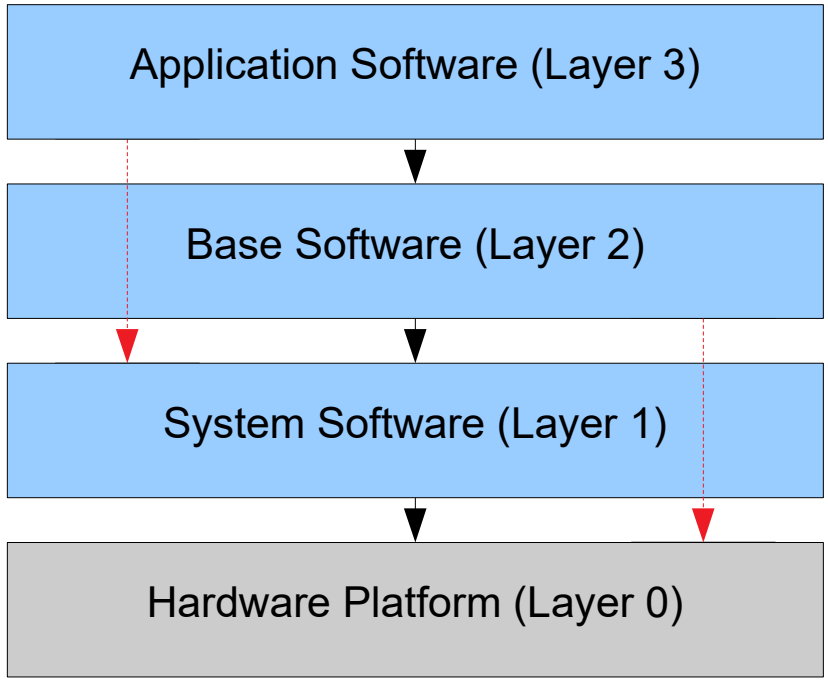
- Distributed Objects
  - OMG CORBA → IIOP, ORB is bus, “Naming Service” is directory
  - MS DCOM → DCE/RPC, “Registry Service” is directory
- Distributed Services
  - Web Services (HTTP, “UDDI” is directory)
    - » UDDI = Universal Description, Discovery, and Integration
  - Sun Jini, now Apache River (RMI/JERI, “Lookup Service” is directory)
  - Enterprise Service Bus
    - Mule, Apache ServiceMix, WSO2, …
  - Message-Oriented Middleware
    - Apache Camel, Apache Kafka

Layers (Abstract Machines)

All problems in computer science can be solved by another level of indirection

D. Wheeler

- Structural pattern
  - Components – layers (abstract machines)
  - Connectors – procedure calls
  - Topology – linear
- Characteristics
  - Constraints – strict layering (limiting interactions only to adjacent layers)



Layers (Abstract Machines) – Evaluation

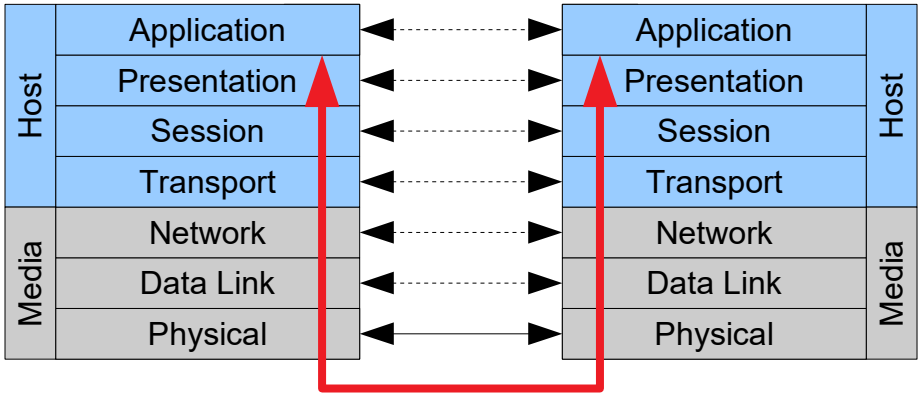
Software Architecture
Shaw & Clements

- Advantages
  - Problem partitioning – supports design based on increasing levels of abstraction (allows implementors to partition a complex problem into a sequence of incremental steps)
  - Modifiability and Portability – because each layer interacts with at most the layers below and above, changes to the function of one layer affect at most two other layers (layers can evolve independently)
  - Reuse – different implementations of the same layer can be used interchangeably, provided they support the same interfaces to their adjacent layers (this leads to the possibility of defining standard layer interfaces to which different implementors can build)
- Disadvantages
  - Not all systems are easily structured in a layered fashion
  - Problem partitioning – it can be quite difficult to find the right levels of abstraction
  - Coupling – considerations of performance may require closer coupling between logically high-level functions and their lower-level implementations
  - Reuse – due to rich interactions between layers it is difficult to define system-independent layers

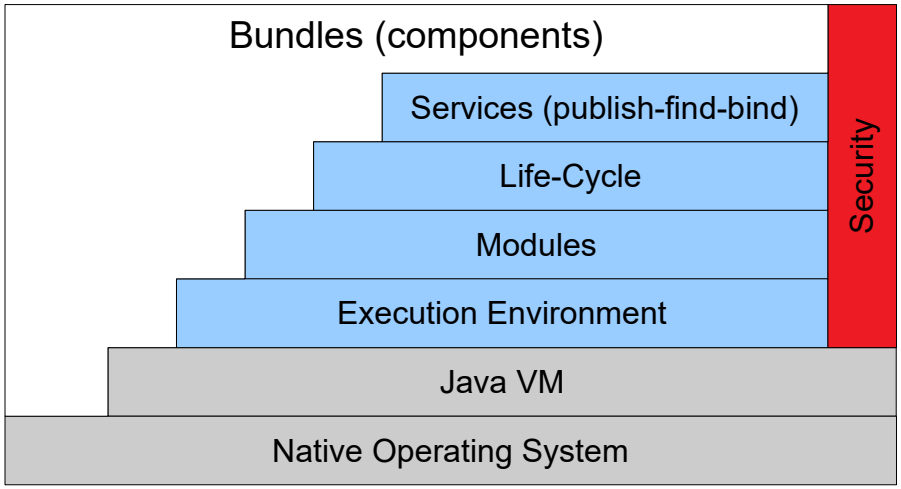
Specializations of Layers (Abstract Machines)

Software Architecture
Shaw & Clements

- Strict layering – OSI (Open Systems Interconnection reference model)



- Non-strict layering and “vertical” (cross-cutting layers) – OSGi (Open Services Gateway initiative) a modular service platform for Java

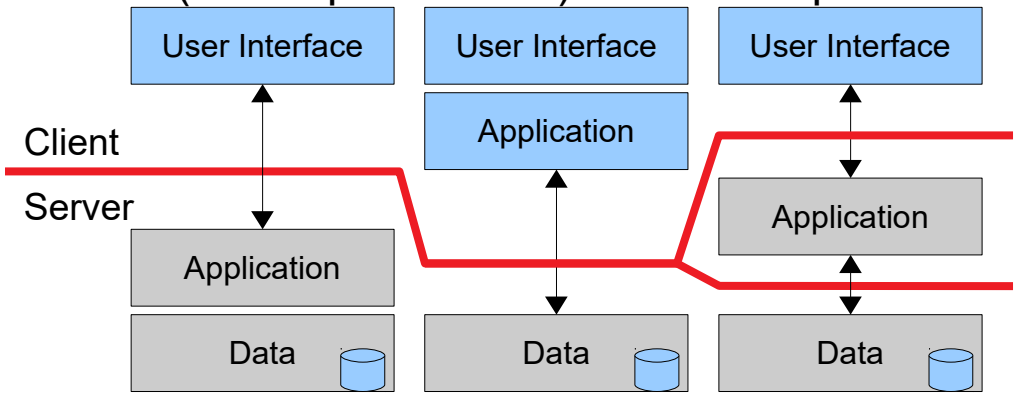


Implementation of “Independent Components” style, using “Layers” style !

Layers (Abstract Machines) Examples

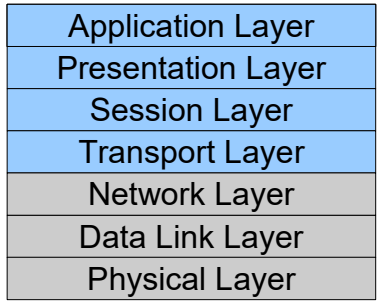
Software Architecture
Shaw & Clements

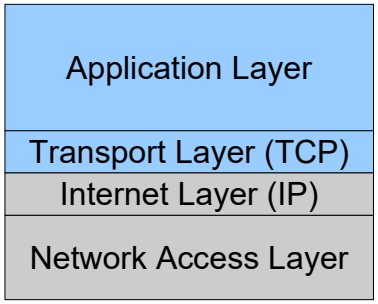
- Multi-tier (layered) systems
  - in module structure (development time) and in component-and-connector structure (run-time)



- Communication stacks

OSI





TCP/IP

- Operating systems, database systems, ...

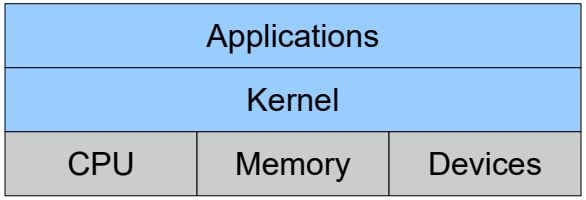


Table-Driven Interpreters (Virtual Machines)

- Structural pattern
  - Components – interpreter (interpretation engine), program memory (containing the program to be interpreted), data or activation record (program state), execution/control state (of the interpreter)
  - Connectors – data and instruction flows
- Characteristics
  - Computational model – various: Turing-machine (imperative), finite-state machines (automata), λ-calculus (functional), ...

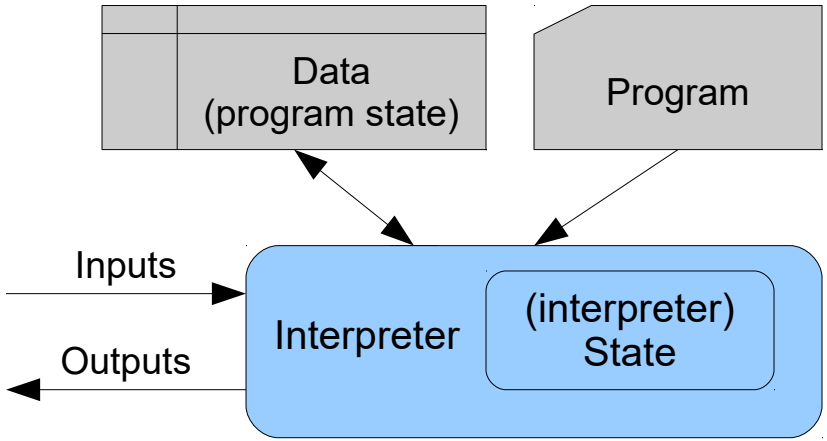


Table-Driven Interpreters (Virtual Machines) – Evaluation

Software Architecture
Shaw & Clements

- Advantages
  - Problem partitioning – supports design based on increased level of abstraction (allows to partition a complex problem into a problem-specific program and suitable abstract machine)
  - Flexibility – highly dynamic behavior through change of program
  - Portability – interpreter can be moved to a different platform without changing the programs
  - Mobility – program can be moved to suitable execution environment
    - code-on-demand – program is requested from remote source and executed locally (e.g. JavaScript or mobile agents)
    - remote execution – program is pushed to remote target (e.g. grid and coud computing)
    - mobile agent – program and some data are moved to different hosts
- Disadvantages
  - Performance – interpretation overhead (partially remedied by “Just-in-Time compiling”)
  - Complexity – defects/bugs can be both in program and in interpreter (requires its own programming environment)
  - Complex memory management
  - Security – mobile program provides attack vectors (can be solved by code signing and/or sandboxing)

Table-Driven Interpreters (Virtual Machines)

Software Architecture
Shaw & Clements

- Specializations
  - byte-code, threaded code, just-in-time compilation, ...
- Examples
  - Excel macros
  - Java Virtual Machine
  - JavaScript
  - Forth (used in Open Firmware)

Some other Familiar Architecture Styles

Software Architecture
Shaw & Clements

- **Main program/subroutine** – main program acts as the driver for the subroutines, typically providing a control loop
- **Distributed (Cooperating) Processes** – independent components characterized by topological features (e.g. ring, star, network, …) or inter-process communication (e.g. one-way, request/reply, heartbeat, broadcast, token, ...)
- **Domain-specific software architectures** (or “reference” architectures for a domain) – to increase the descriptive power of structures (in many cases an executable system can be generated from the architectural description itself)
- **State transition systems** – a common organization for many reactive systems (defined in terms of a set of states and a set of named transitions that move a system from one state to another)
- **Process control systems** – systems intended to provide dynamic control of a physical environment (roughly characterized as a feedback loop in which inputs from sensors are used by the process control system to determine a set of outputs that will produce a new state of the environment)

Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

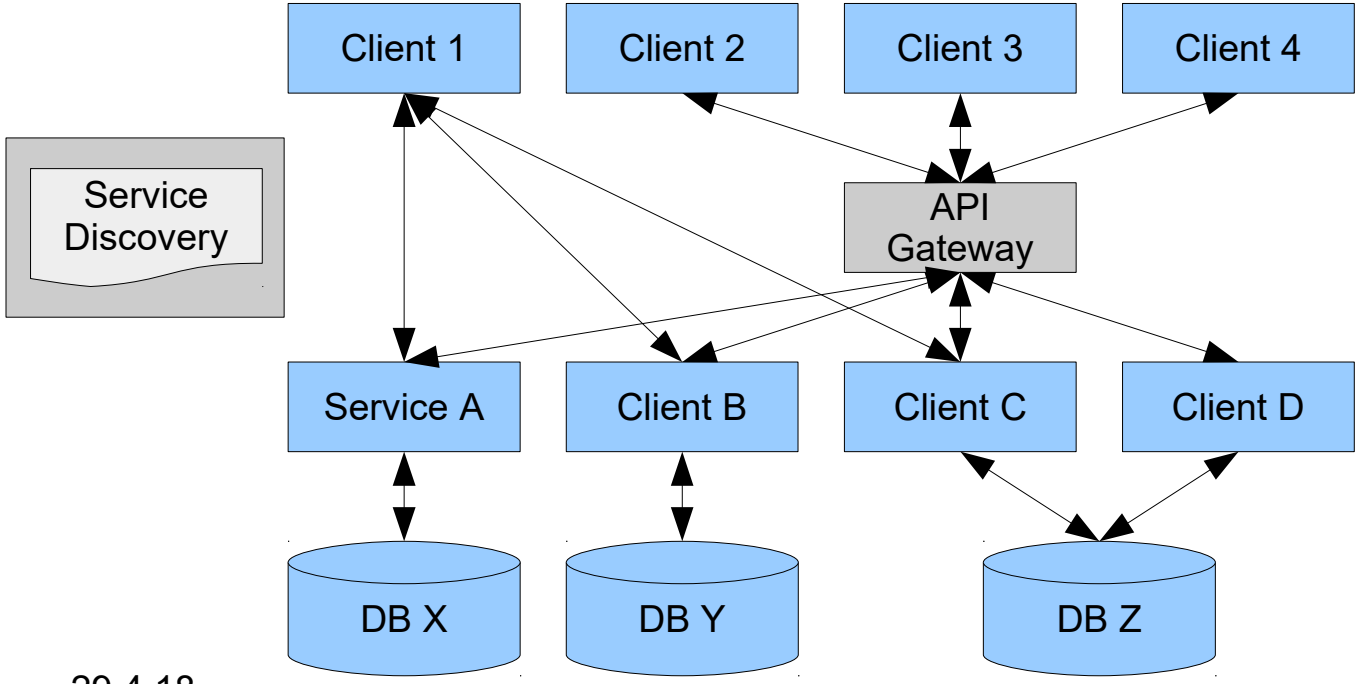
Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Micro-Services

A new and “better” SOA ?

- Structural pattern
  - Components – clients, micro-services, data-stores, opt. API gateway, opt. service discovery
  - Connectors – service requests
  - Topology – arbitrary or star (when using API gateway)
- Characteristics
  - Constraints – services are small independent and loosely coupled (often organized around business capabilities)



Distributed computing fallacies:
1.The network is reliable
2.Latency is zero
3.Bandwidth is infinite
4.The network is secure.
5.Topology doesn't change
6.There is one administrator
7.Transport cost is zero
8.The network is homogeneous

P. Deutch & J. Gosling

Micro-Services – Evaluation

A new and “better” SOA ?

- Advantages
  - Robustness – improved fault isolation (applications can remain unaffected by the failure of a single module)
  - Scalability – single services can be scaled by cloning or by data partitioning
  - Portability – eliminated long-term commitment to a single technology stack
  - Changeability – independent deployment and rolling back changes are much easier
  - Undertandability – makes it easier for a new developer to understand the functionality of a service
- Disadvantages
  - Complexity of development – distributed systems need extra functionality to handle errors/disruptions and latency
  - Complexity of deployment and operations
    - multiple databases and transaction management can be difficult and take large effort
    - deployment of micro-services may require coordination among multiple services, which may not be as straightforward as deploying a monolith in a container
  - Testability – testing a micro-services-based application can be cumbersome (each dependent service needs to be confirmed before you can start testing)

Micro-Services vs. Service Oriented Architecture (SOA)

A different architecture style ?

- Micro-Services
  - For modularization of applications into loosely coupled components
  - No centralized service management
  - Focuses on decoupling (decomposition of application)
  - No centralized communication infrastructure (better error tolerance)
  - Usually single simple communication protocol
  - Limits integration choices
  - Usually smaller granularity of components
  - Multiple independent data-stores
  - Relaxed governance
  - Better suited **for compact and well-partitioned applications**
- Service-Oriented Architecture (SOA)
  - For composition of application from independent components
  - Centralized service management (Enterprise Service Bus)
  - Focuses on reuse (of business functionality)
  - Usually centralized communication infrastructure (single point of failure)
  - Supports multiple communication protocols
  - Focuses on interoperability
  - Usually larger granularity of components
  - Usually share data-store
  - Common governance
  - Better suited **for large complex enterprise applications (sets of applications)**

Micro-Services Examples

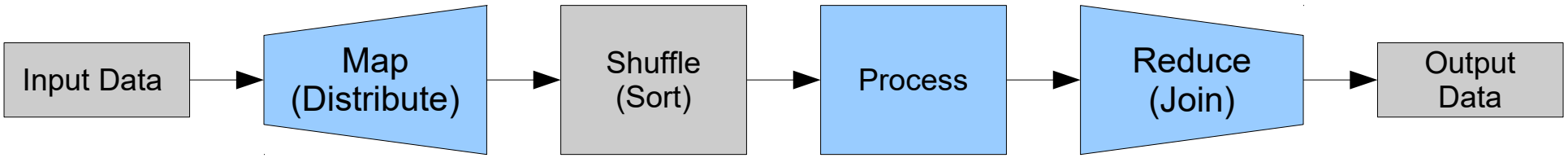
A new and “better” SOA ?

- Specializations
  - with API Gateway
  - with light-weight integration bus
  - with service mesh – kind of “distributed (integration) bus”
- Examples
  - systems of many large web companies (Netflix, eBay, Amazon, Twitter, PayPal, ...)
  - tools/frameworks
    - Spring Boot
    - Jersey (for REST)

Map-Reduce Architecture

dīvide et imperā

- Structural pattern
  - Components – mapper, opt. shuffle, reducer, processors
  - Connectors – data flows
  - Constraints –
- Characteristics
  - Computational model –
  - Invariants – must consist of exactly one Map function followed by an optional Reduce function
  - Theory – λ-calculus (A. Church), higher-order functions or functors (map, filter, fold/reduce)



Map-Reduce Architecture – Evaluation

dīvide et imperā

- Advantages
  - Scalable – efficiently execute programs on large clusters, by exploiting data parallelism
  - Simple – parallelization complexity is handled by the framework
  - Portability – independent of the underlying storage mechanism
- Disadvantages
  - Inflexible – each job must consist of exactly one Map followed by an optional Reduce, and steps cannot be executed in a different order or overlapped
  - Performance highly depends on the nature of the application
  - Doesn't support incremental computations

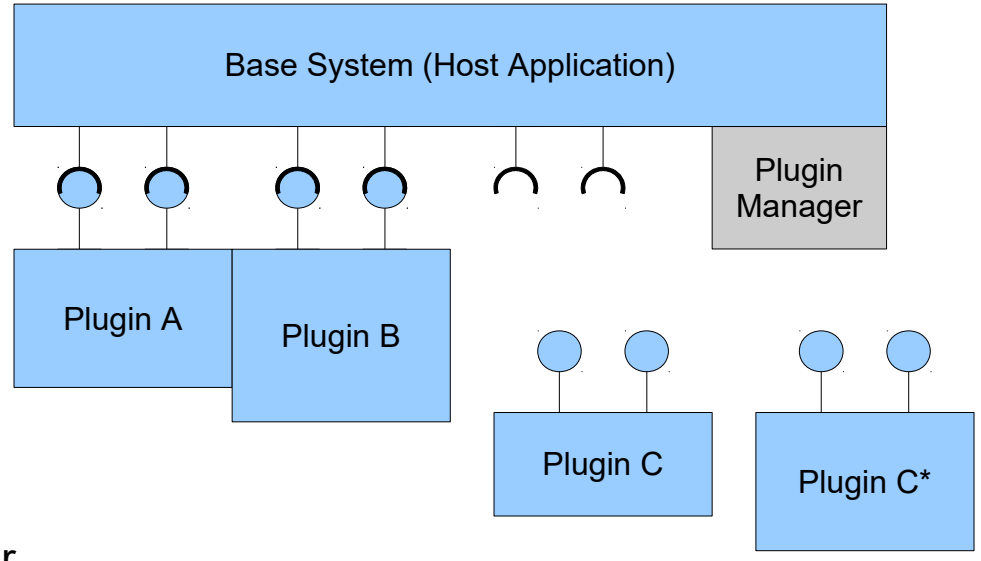
Map-Reduce Architecture Examples

dīvide et imperā

- Specializations
  - Iterative Map-Reduce (e.g. Google's PageRank)
- Examples of its use
  - Apache Hadoop (MapReduce)
  - Twister (iterative map-reduce)
  - applied for
    - inverted index construction
    - document clustering
    - machine learning
    - ...

Plugin Architecture Style

- Structural pattern
  - Components – base system, plugins, opt. plugin manager
  - Connectors – service interfaces
  - Topology – star
- Characteristics
  - Constraints – plugins conform to service interfaces and depend on base system services



- Examples
  - Mozilla browser
  - Eclipse IDE

Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

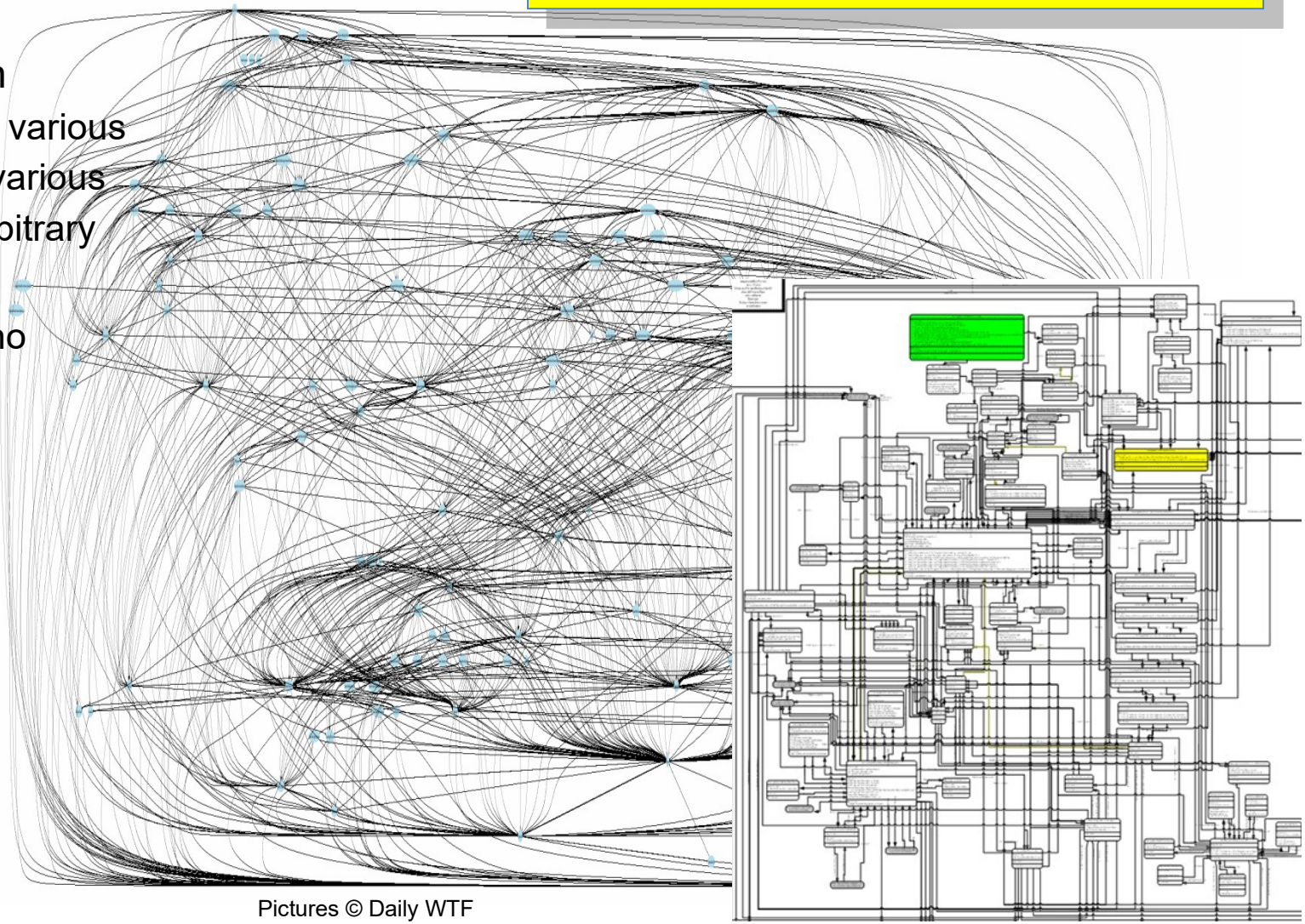
Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Emergent Architecture – a.k.a. “Big Ball of Mud” (B. Foote & J. Yoder)

De-Facto Standard Software Architecture !

- Structural pattern
  - Components – various
  - Connectors – various
  - Topology – arbitrary
- Characteristics
  - Constraints – no
  - Invariants – no
  - Theory – no



Pictures © Daily WTF

Emergent Architecture – a.k.a. “Big Ball of Mud” (B. Foote & J. Yoder)

Complexity increases rapidly until it reaches a level of complexity just beyond that with which we can comfortably cope

W. Cunningham

- Emerges from
  - “Throwaway code” – a.k.a. quick hack, prototype or “spike” in XP/agile
  - “Piecemeal growth” – a.k.a iterative/incremental development
  - “Keep it working” – i.e. overhaul is needed, but you have to keep system working
  - “Shearing layers” – i.e. different artifacts change at different rates
  - “Sweeping it under the rug” – i.e. even if you can’t clean the mess, hide it

- Forces corresponding to emergence
  - *time* – designing architecture takes time
  - *cost* – designed architecture costs and is long-term investment
  - *experience* and *skill* – designing architecture requires know-how
  - *visibility* – software is not tangible, software architecture is “under the hood”
  - *complexity* and *scale* of the problems – software is ugly because the problem is ugly
  - *change* – predicting future change requires vision and courage
  - *organization* – architecture reflects organization

Conway’s law !

Conway's Law

Make sure the organization is compatible with the product architecture

J. Coplien

- **Organizations, produce designs, which are copies of these organizations**

- Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure
  - a design effort should be organized according to the need for communication
  - because the design which occurs first is almost never the best possible – flexibility of organization is important to effective design

(M. Conway 1968)

Effects of Conway's Law

Greatest single common factor behind many poorly designed systems has been the availability of a design organization in need of work

M. Conway

- Why do large systems disintegrate?
  - First, the realization by the initial designers that the system will be large, together with certain pressures in their organization, make irresistible the temptation to assign too many people to a design effort
  - Second, application of the conventional wisdom of management to a large design organization causes its communication structure to disintegrate (due to delegation and Parkinson's law of expanding organizations)
  - Third, the homomorphism of system to the design organization insures that the structure of the system will reflect the disintegration which has occurred in the design organization

    (M. Conway 1968)

“Big Ball of Mud” – Evaluation

De-Facto Standard Software Architecture !

- Advantages
  - Quick to make → fast Time-to-Market
  - Cheap to make → good Cost vs. Benefit ratio
  - Does not need planning, nor governance → it just emerges!
  - Does not need skills → anybody could/would do it

Mostly business concerns !

- Disadvantages
  - Maintainability → difficult to maintain
  - Modifiability → hard to change
  - Testability → difficult to test

Mostly IT concerns !

Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Using Styles in System Design
Choosing Styles to fit the Problem

"Boxology" – Shaw & Clements

- Use **data flow** (batch sequential or pipe-and-filter = pipeline) style if problem
  - can be decomposed into sequential stages (if each stage is incremental, consider pipeline)
  - involves transformations on continuous streams of data (when problem involves passing rich data representations, avoid pipes restricted to ASCII)
- Use **data abstraction** or **object-oriented** style, if
  - representation of data is likely to change over the lifetime of the system
  - a central issue is understanding data, its management, and its representation (if the data is long-lived, use on repositories)
- Use **data-centered** (repository) style if data is long-lived and use
  - blackboard, if the input data is noisy (low signal-to-noise ratio) and the execution order cannot be predetermined
  - database management system, the execution order is determined by a stream of incoming requests and the data is highly structured

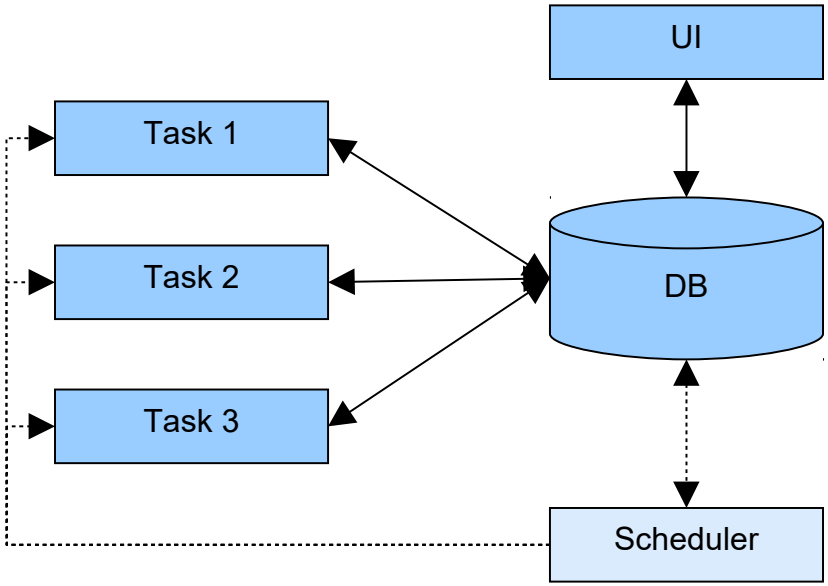
Using Styles in System Design
Choosing Styles to fit the Problem

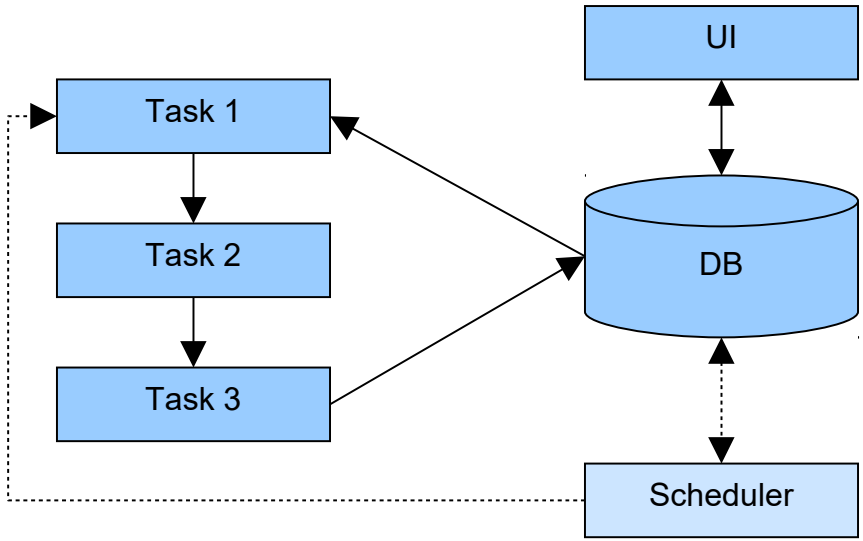
"Boxology" – Shaw & Clements

- Use **closed loop** control architecture, if problem
  - involves controlling continuing action, is embedded in a physical system, and is subject to unpredictable external perturbation
- Use **interpreter**, if problem
  - computational but there's no machine on which it can be executed
- Use of other styles
  - **independent components** or cooperating processes, if task requires a high degree of flexibility/configurability, loose coupling between tasks, and reactive tasks
  - an **event-based implicit invocation** architecture, if there's reason not to bind the recipients of signals from their originators,
  - a *replicated worker* or *heartbeat* style **cooperating processes**, if the tasks are of a hierarchical nature
  - **client-server**, if the tasks are divided between producers and consumers
  - a *token passing* style **cooperating processes**, if it makes sense for all of the tasks to communicate with each other in a fully connected graph

Example

Different Architecture Styles → Different Properties

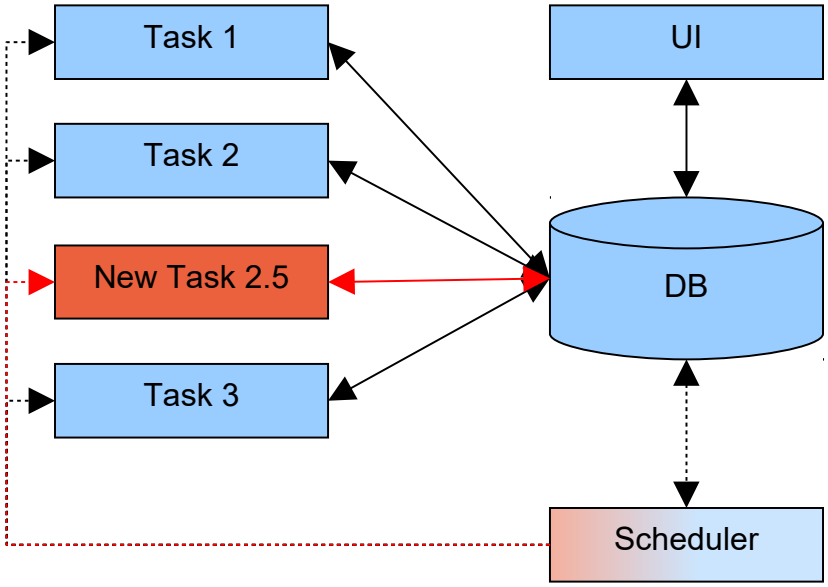


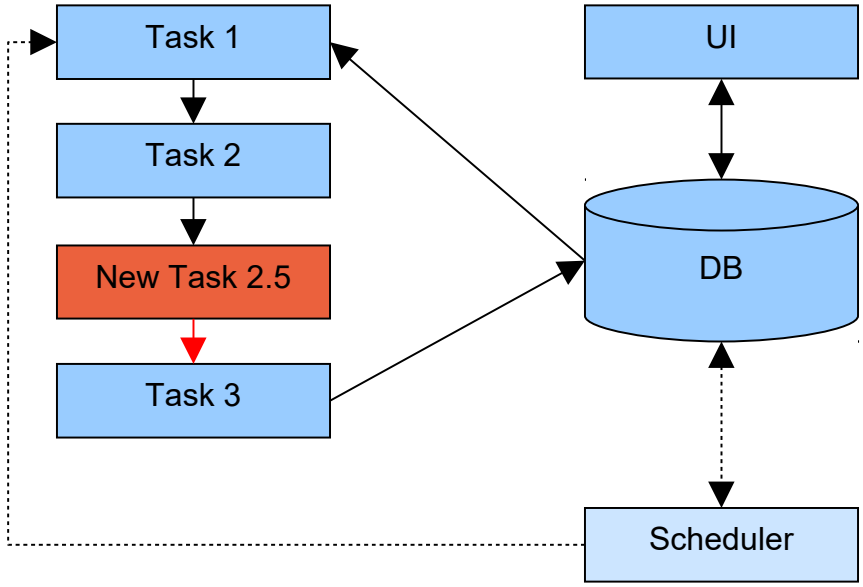


Example

Different Architecture Styles → Different Properties

adding new task

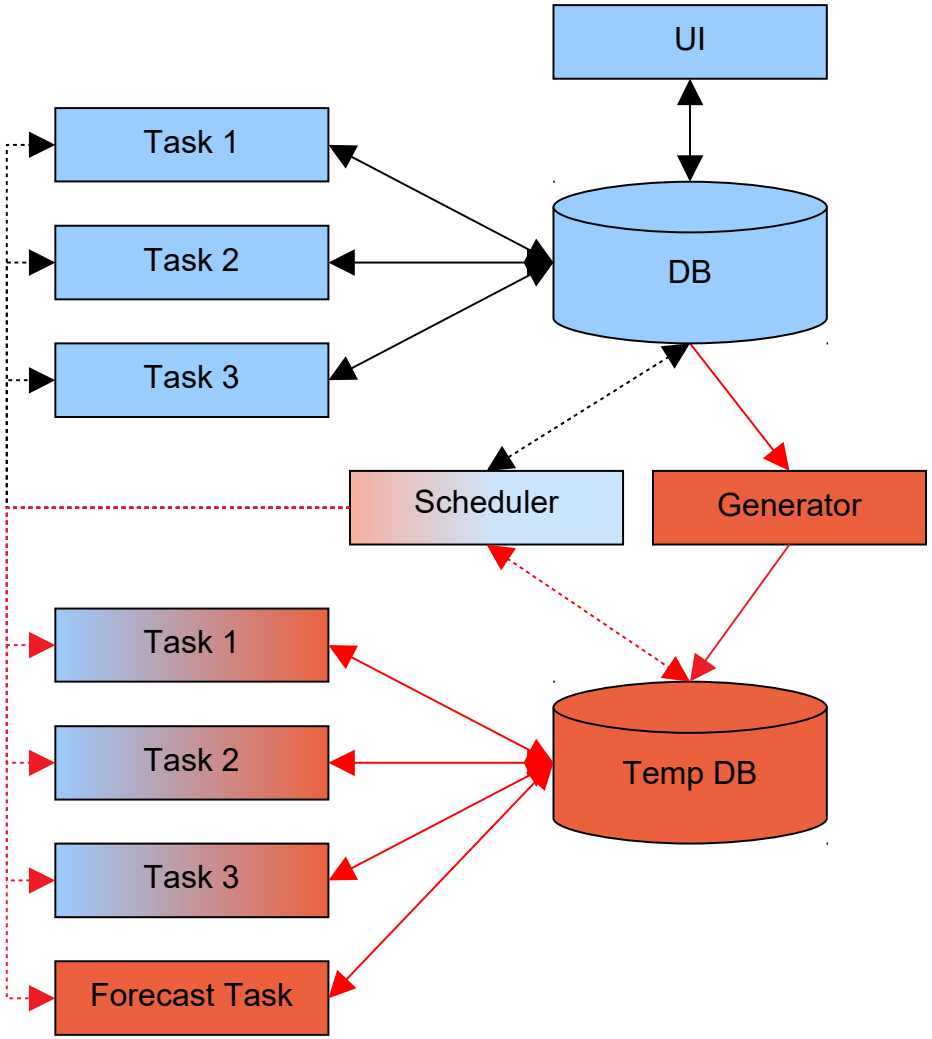


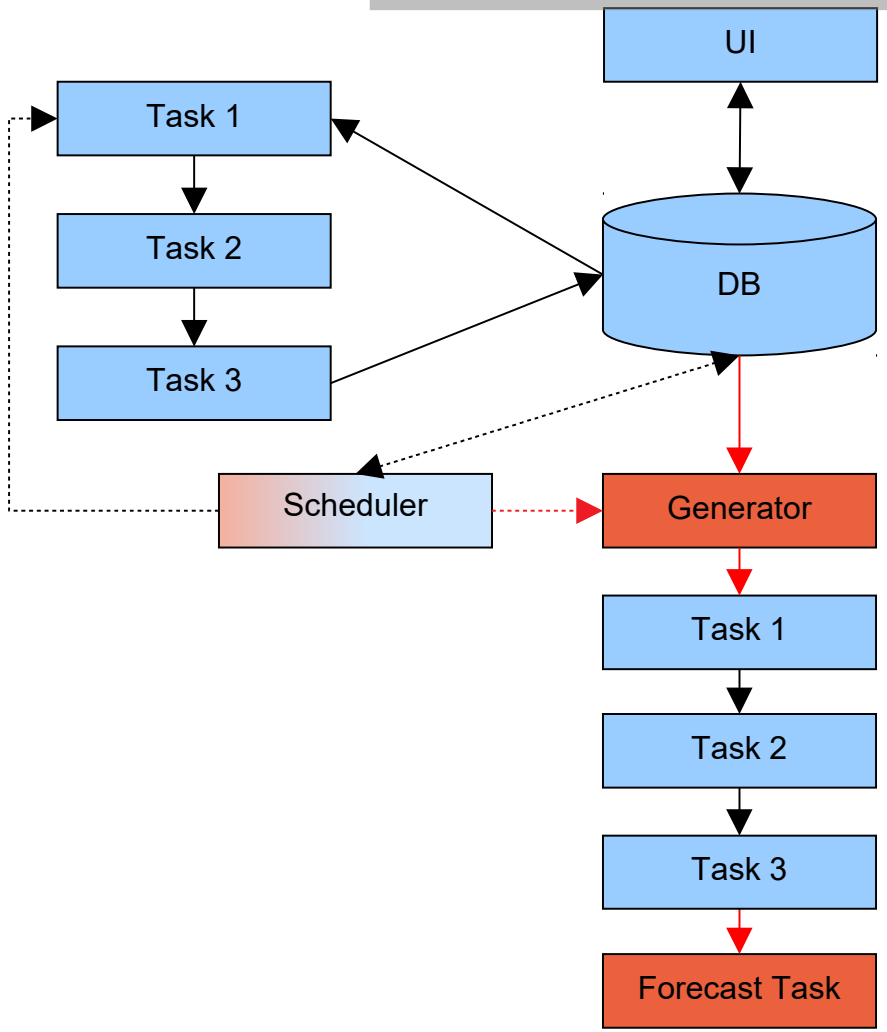


Example

Different Architecture Styles → Different Properties

adding forecasts of portfolio





Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Different ways to combine Architecture Styles

Software Architecture
Shaw & Clements

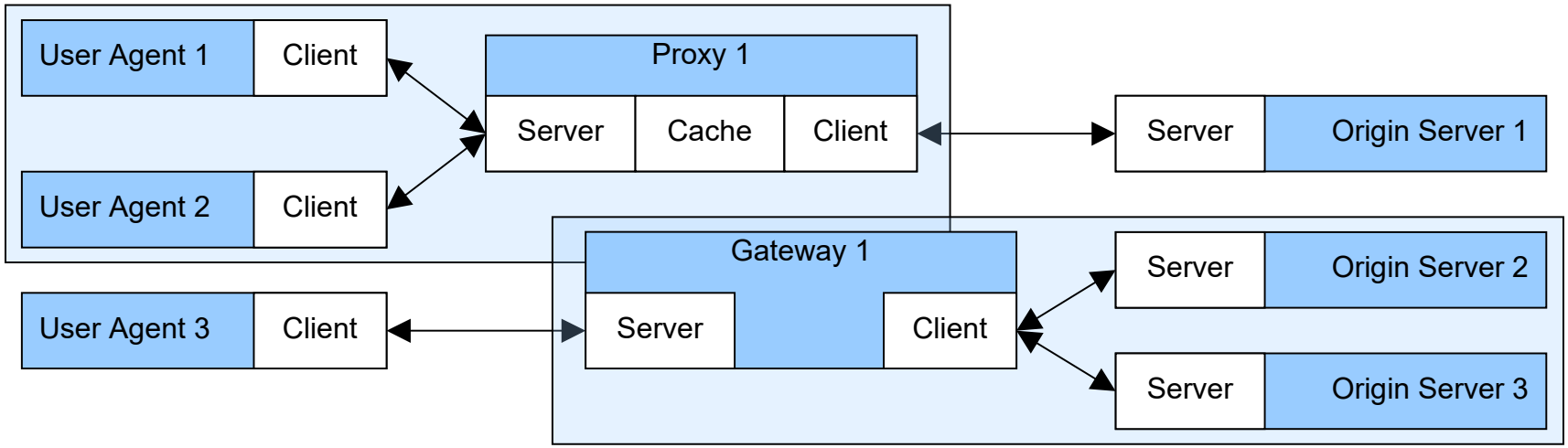
- **Mixing** architecture styles – using elements from many architecture styles together
- **Hierarchical decomposition** – an element of a system organized in one architectural style may have an internal structure that is developed a completely different style
- **Conforming** same architecture element (component or connector) **to multiple** architecture styles – having multiple architectural connectors for a single component or combining multiple component types for a given architecture element
  - access a repository through part of its interface, but interact through pipes with other components in a system, and accept control information through another part of its interface
  - “active database” – a repository which activates external components through implicit invocation (e.g. blackboards are often constructed this way)

REpresentational State Transfer (REST)

(Compound Style)

architecture of web !

- Structural pattern
  - Components
    - Data (resources, resource identifiers, representations, representation metadata, resource metadata, control data)
    - Processing (origin servers, gateways, proxies, user agents)
  - Connectors – clients, servers, caches, resolvers, tunnels
- Characteristics
  - Constraints – data is not encapsulated
  - Theory – Fielding analysis



REpresentational State Transfer (REST) – Evaluation

architecture of web !

- Advantages
  - Simplicity – no need for explicit resource discovery mechanism (due to hyper-linking) and uniform interface
  - Scalability – stateless communication, layered system
  - Efficiency – caching promotes network efficiency and fast response times
  - Evolvability – support of document type evolution (such as HTML and XML) without impacting backward or forward compatibility
  - Extensibility – allows support for new content types without impacting existing and legacy content types
- Disadvantages
  - Limited functionality – selected uniform interface (HTTP) is difficult for handling real time asynchronous events
  - Scalability – managing URI namespace can be cumbersome, this can impact network performance by encouraging more frequent client-server requests and responses
  - Visibility – in case code-on-demand is used to extend the client

Representational State Transfer (REST) Examples

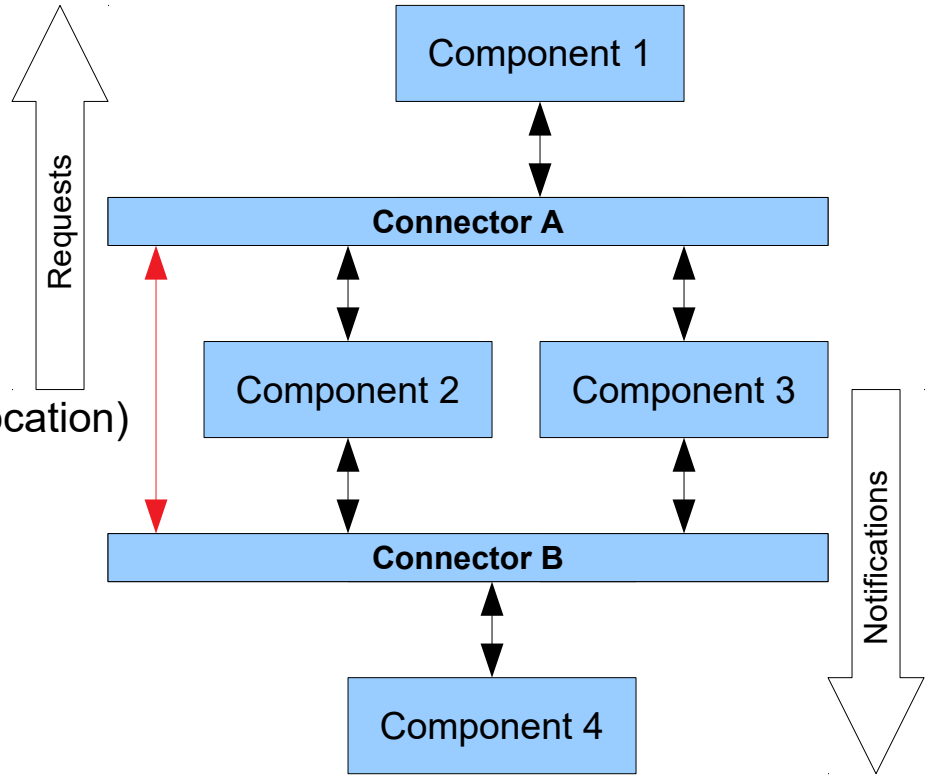
architecture of web !

- WWW (World Wide Web)
- CMIP/CMOT (Common Management Information Protocol)
- Amazon Web Services (AWS) REST API
- IBM WebSphere Portal REST API

Chiron-2 (C2) Architecture

a component called connector !

- Structural pattern
  - Components – components (have state & behavior), connectors (message routing devices)
  - Connectors – communication links (for messages)
  - Topology – hierarchical network
- Characteristics
  - Constraints
    - each component and connector has
      - top (which emits requests) and
      - bottom (which emits notifications)
    - components and connectors are organized into layers (strata)
  - Computational model – event-based (implicit invocation)
  - Invariant
    - components may have own thread(s) of control
    - limited visibility (substrate independence)
    - message based communication
    - no shared address space
  - Theory – UCI Chiron-2 formalism (in Z)



Chiron-2 (C2) Architecture – Evaluation

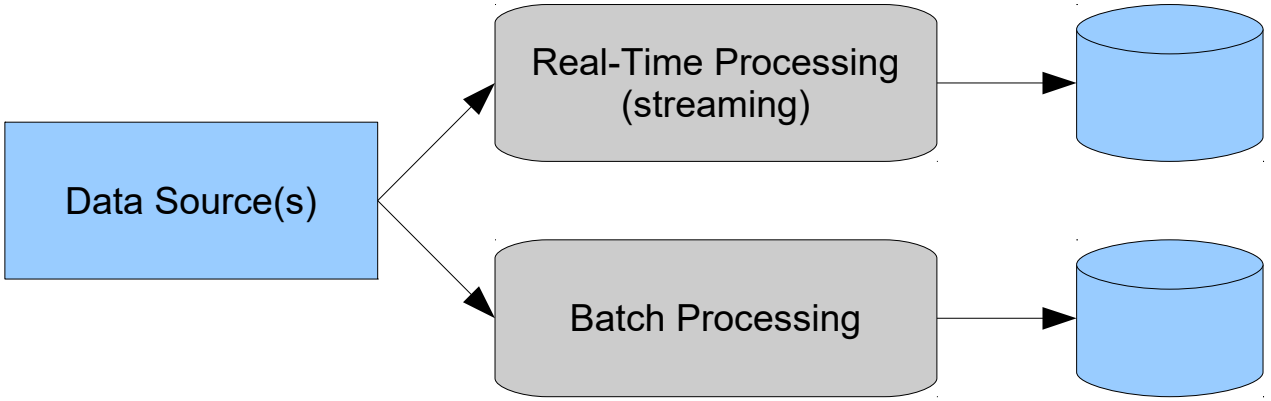
a component called connector !

- Advantages (according to UCI SRI)
  - Separation of Concerns – encourages modular strategy
  - Scalability – supports multiple levels of component interface granularity
  - Extensibility – limiting component interdependence (components have standardized interfaces)
  - Flexibility/Modifiability – by incorporating additional or re-configuring existing components (prior to or during execution)
  - Reliability – components can be carefully designed, implemented, and verified before usage
  - Cost Reduction – component reuse and architectural guidance
  - Understandability – the use of high level models
  - Distributability/Parallelization – no assumption of shared memory or address space

- Disadvantages (according to Fielding)
  - Scalability/Efficiency – does not support intermediaries/caching (no generic resource interface, no guaranteed stateless interactions)

Lambda Architecture

- Structural pattern
  - Components – data sources, batch data processors (providing accuracy), streaming data processors (filling gap due to batch processors lag), query service/servers
  - Connectors – data channels
- Characteristics
  - Constraints – depends on a data model with an append-only, immutable data source that serves as a system of record
  - Theory –



Lambda Architecture – Evaluation

- Advantages
  - balances latency, throughput, and fault-tolerance by using batch processing to provide comprehensive and accurate views of batch data, while simultaneously using real-time stream processing to provide views of online data
- Disadvantages
  - inherent complexity – different code base needed for streaming and batch side must be maintained and kept in sync
- Examples
  - Yahoo
  - Netflix Suro

Content

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

> Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction
>
> Roy Fielding

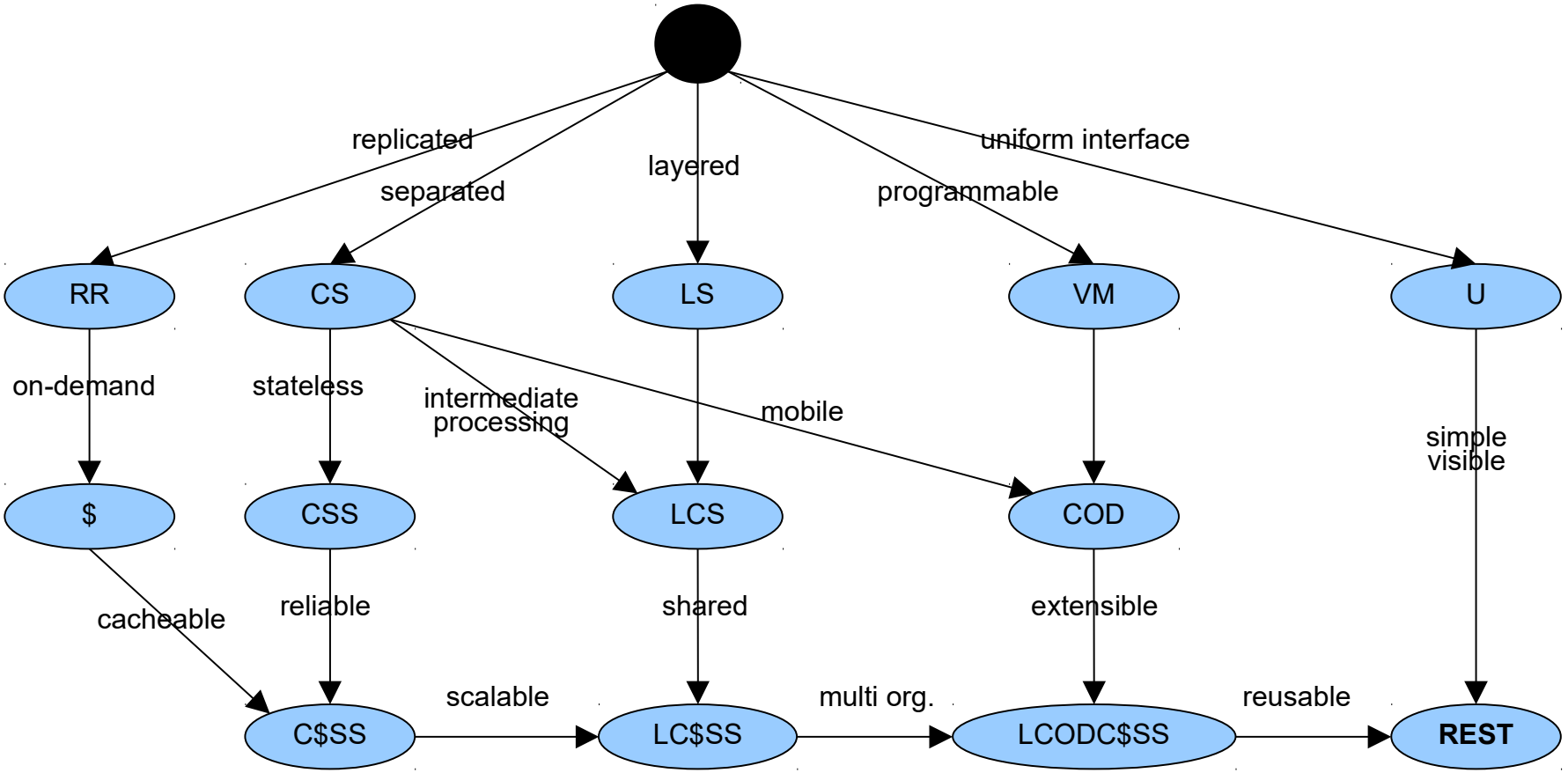
Representational State Transfer (REST)
Constituent Styles

architecture of web !

Architecture Style		Desired Properties
Null Style	an empty set of constraints	
Client-Server Style (CS)	separation of concerns	modifiability, independent evolution
Stateless Communication (S)	session state in client	visibility, reliability, scalability
Cache ($)	a variant of Replicated Repository (RR)	network efficiency
Uniform Interface (U)	a constrained set of well defined operations and content types	simplicity, portability
Layered System Style (LS)	hierarchical decomposition, managing complexity	simplicity, scalability
{optional} Code-on-Demand (COD)	based on Virtual Machine – simplified clients, but lower visibility	modifiability (extensibility), simplicity

Deriving REST from Constituents

architecture of web !



Content

Each (architecture) style provides an abstraction for the interactions of components, capturing the essence of a pattern of interaction

Roy Fielding

- Intro
  - Different Levels of Commonality in Software
- Software Architecture Style
- Classifications of Software Architecture Styles
- Analysis of some Software Architecture Styles
  - Main styles: data flow, call and return, data centered, …
  - Some “modern” styles: micro-services, map-reduce, …
  - Emergent architecture – “Big Ball of Mud”
- Using Software Architecture Styles in Design
- Derived (Complex) architecture styles (REST, ...)
- Designing an Architecture Style (on example of REST)
- Conclusions

Conclusions

A coherent package of pre-made design decisions that provide a set of properties

- Architectural **structures** can embody decisions how the system
  - is to be structured as a set of code or data **units that have to be constructed or procured**
  - is to be structured as a set of **elements that have run-time behavior** – (components) and interactions (connectors)
  - will **relate to non-software structures in its environment**

- **Architecture Style**
  - characterizes **a family or a class** of system architectures that are related by shared structural and semantic properties
  - is defined by
    - a **vocabulary of design elements**
    - **design rules**, or constraints (incl. topology)
    - **semantic interpretation**
    - **analyses** that can be performed on systems built in that style

Conclusions

A coherent package of pre-made design decisions that provide a set of properties

- Usage of Architecture Styles Supports
  - ***Design Reuse*** – *w*ell-understood solutions can be applied to new problems
  - ***Code Reuse*** – shared implementations of invariant aspects of a style
  - ***Understandability of System Organization*** – e.g. meaning of “client-server”
  - ***Interoperability*** – supported by style standardization
  - ***Style-Specific Analysis*** – enabled by the constrained design space
  - ***Visualizations*** – style-specific descriptions matching engineer’s mental models (e.g. stack diagrams for layers)

- Main Architecture Styles can and must be combined
  - to achieve the required properties of interest
  - to match the problem structures (e.g. ways of decomposition) or problem nature
  - by mixing styles, using hierarchical decomposition or conforming architecture elements to multiple styles

53. The great way is easy, yet programmers prefer the side paths. Be aware when things are out of balance. Remain centered within the design.

Lao Tsu (by Philippe Kruchten)

Thank You!

Questions

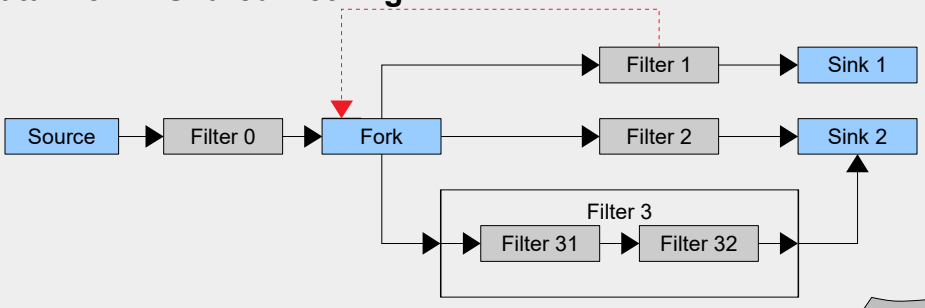
- List ways of representing commonalities of design on different levels?
- What is software architecture style?
- What are the main parts of an architecture style?
- Describe classification principles of software architecture styles?
- List the main software architecture styles?
- Describe named software architecture style, what are its structural pattern, advantages and disadvantages?
- Describe the benefits of architecture styles?
- Formulate Conway's law, describe how is affects software architecture?
- How can be different architecture styles combined?
- What kinds of structures are present in software architecture, and what decisions these structures embody?
- Describe REST style and rationale behind its constituent architecture styles?
- What are constituent styles of C2?
- How to choose the suitable architecture style
- What architecture styles support best
  - scalability?
  - portability & modifiability?

Literature

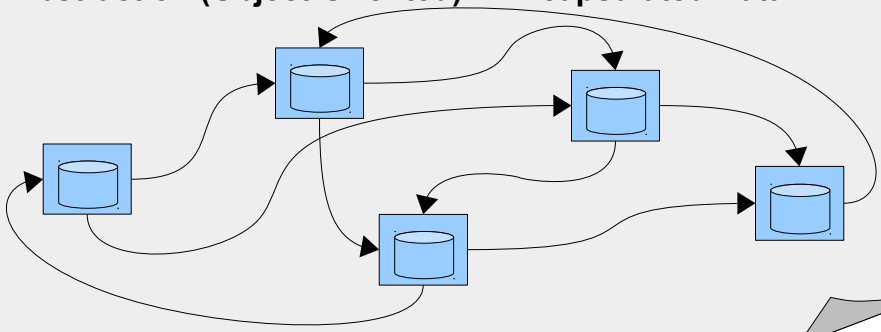
- Different Architecture Styles
  - https://www.cs.cmu.edu/afs/cs/project/vit/ftp/pdf/Boxology.pdf
  - https://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf
  - https://msdn.microsoft.com/en-us/library/ee658117.aspx
  - https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013
- Particular Architecture Styles
  - https://martinfowler.com/articles/microservices.html
  - http://www.laputan.org/mud/
  - http://www.melconway.com/Home/Conways_Law.html
  - http://www.ics.uci.edu/~arcadia/C2/c2.html
  - https://research.google.com/archive/mapreduce.html
- Constructing Software Style
  - https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm
- … Google “software architecture style” …

Main Architecture Styles

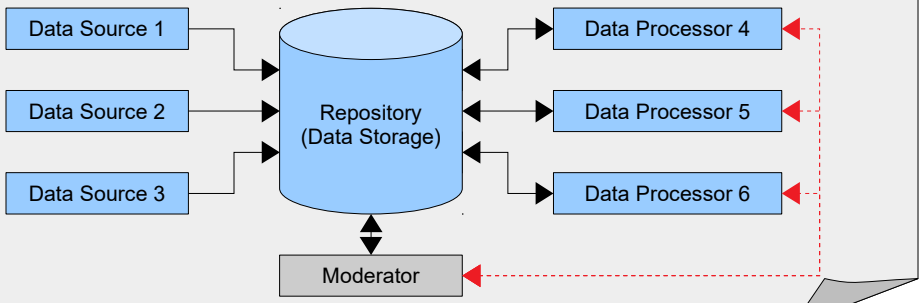
Data-Flow = Shared Nothing



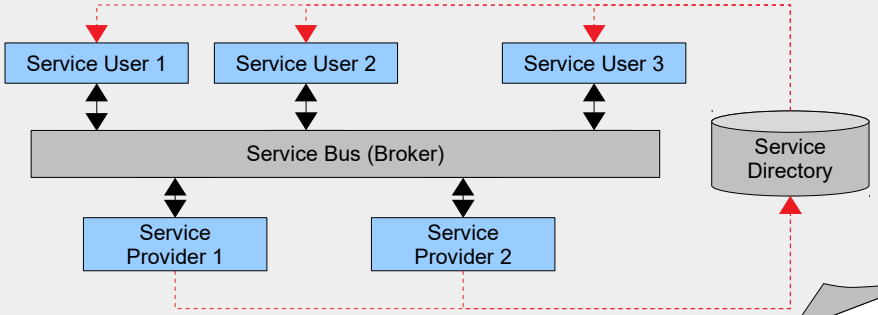
Data Abstraction (Object-Oriented) = Encapsulated Data



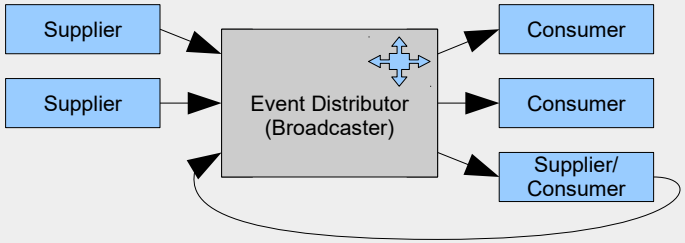
Data-Centered = Shared Everything



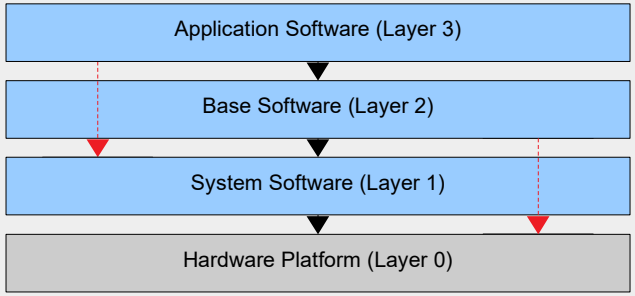
Independent Components = Interoperability



Event-Based (Implicit Invocation) = Decoupled Control



Layers (Abstract Machines) = Portability



Classification of Architectural Styles

"Boxology" – Shaw & Clements

Style	Constituent Parts		Control Issues		Data Issues		Control/Data Interaction
	Components	Connectors	Topology	Synchronicity	Topology	Continuity	Isomorphic
Data Flow Architectural Styles							
Batch Sequential	programs	data batches	linear	sequential	linear	sporadic	yes
Data Flow Network	transducers	data streams	arbitrary	asynchronous	arbitrary	continuous	yes
Pipes and Filters	filters	pipes	linear	asynchronous	linear	continuous	yes
Call and Return Architectural Styles							
Main Program/Subroutines	procedures	static calls	hierarchical	sequential	arbitrary	sporadic	no
Abstract Data Types	managers	static calls	arbitrary	sequential	arbitrary	sporadic	yes
Objects	managers	dynamic calls	arbitrary	sequential	arbitrary	sporadic	yes
Call-based Client Server	programs	remote calls	star	synchronous	star	sporadic	yes
Layered		calls	hierarchical	any	hierarchical	sporadic	often
Independent Components Architectural Styles							
Event Systems	processes	signals	arbitrary	asynchronous	arbitrary	sporadic	yes
Communicating Processes	processes	messages	arbitrary	non-sequential	arbitrary	sporadic	possibly
Data Centered Architectural Styles							
Repository	memory, computations	queries	star	asynchronous	star	sporadic	possibly
Black-Board	memory, components	direct access	star	asynchronous	star	sporadic	no

Software Architecture
Kinds of Structures

CMU SEI

Kind	Structure	Elements	Relations	Decisions	Quality Attribute
Module Structures	Decomposition	Module	sub-module-of	Decomposition, structuring, information hiding, encapsulation	Modifiability
	Uses	Module	uses (requires)	Usable/useful sub-sets, extensions	Extensibility, Subsetability
	Layers	Layer	uses (requires), provides abstraction	Portability, ease of change and abstraction “virtual machines”	Portability
	Class	Class, Object	is-a (specializes), instance-of, ...	Reuse, commonality and planned incremental extension	Modifiability, Extensibility
	Data Model	Data Entity	{one,many}-to-{one,many}	Global data structures consistency	Modifiability, Performance
Component & Connector Structures	Service	Service, Bus, Registry, ...	runs-concurrently, excludes, precedes, ...	Independent development of components	Interopertability, Modifiability
	Concurrency	Process, Thread	can-run-parallel	Parallelism, access to resources	Performance, Availability
Allocation Structures	Deployment	Component, Hardware Devices, ...	allocated-to, migrates-to	Performance, security, availability	Performance, Security, Availability
	Implementation	Module, File Structure, ...	stored-in	Development, integration and testing	Development Efficiency
	Work Assignment	Module, Organization Unit, ...	assigned-to	Project management and communication	Development Efficiency

Using Architecture Styles (R. N. Taylor, N. Medvidovic, E. M. Dashofy)

Architecture Style	Summary	Use it when	Avoid it when
Call-and-Return			
Main program & subroutines	Main program controls program execution, calling multiple subroutines	Application is small and simple	Complex data structures needed Future modifications likely
Object-Oriented	Objects encapsulate state and accessing functions	Close mapping between external entities and internal objects is sensible Many complex and interrelated data structures	Application is distributed in a heterogeneous network Strong independence between components necessary High performance required
Layered (Tiered)			
Abstract Machines	Virtual machine, or a layer, offers services to layers above it	Many applications can be based upon a single, common layer of services Interface service specification resilient when implementation of a layer must change	Many levels are required (causes inefficiency) Data structures must be accessed from multiple layers
Client-Server	Clients request service from a server	Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation	Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine cap abilities rival or exceed the server's
Data-Flow			
Batch Sequential	Separate programs executed sequentially, with batched input	Problem easily formulated as a set of sequential, severable steps	Interactivity or concurrency between components necessary or desirable Random-access to data required
Pipe-and-Filter	Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters	As with batch-sequential, filters are useful in more than one application Data structures easily serializable	Interaction between components required Exchange of complex data structures between components required

Using Architecture Styles (R. N. Taylor, N. Medvidovic, E. M. Dashofy)

Architecture Style	Summary	Use it when	Avoid it when
Data-Centered (Shared Memory)			
Blackboard	Independent programs, access and communicate exclusively through a global repository known as backboard	All calculation centers on a common, changing data structure; order of processing dynamically determined and data-driven	Programs deal with independent parts of the common data Interface to common data susceptible to change When interactions between the independent programs require complex regulation
Virtual Machines			
Interpreter	Interpreter parses and executes the input stream, updating the state maintained by the interpreter	Highly dynamic behavior required High degree of end-user customizability	High performance required
Mobile Code	Code is mobile, that is, it is executed in a remote host	When it is more efficient to move processing to a data-set than the data-set to processing When it is desirable to dynamically customize a local processing node through inclusion of external code	Security of mobile code cannot be assured,or sand-boxed When tight control of versions of deployed software is required
Implicit Invocation			
Publish-Subscribe	Publishers broadcast messages to subscribers	Components are very loosely coupled Subscription data is small and efficiently transported	When middle-ware to support high-volume data is unavailable
Event-Based	Independent components aynchronously emit and receive events communicated over event buses	Components are concurrent and independent Components heterogeneous and network-distributed	Guarantees on real-time processing of events is required

Using Architecture Styles (R. N. Taylor, N. Medvidovic, E. M. Dashofy)

Architecture Style	Summary	Use it when	Avoid it when
Peer-to-Peer			
Peer-to-peer	Peers hold state and behavior and can act as both clients and servers	Peers are distributed in a network, can be heterogeneous and mutually independent Robust in face of independent failures Highly scalable	Trustworthiness of independent peers cannot be assured or managed Resource discovery inefficient without designated nodes
Complex Styles			
C2	Layered network of concurrent components communicating by events	When independence from substrate technologies required Heterogeneous applications When support for product-lines desired	When high-performance across many layers required When multiple threads are inefficient
Distributed Objects	Objects instantiated on different hosts	Objective is to preserve illusion of location-transparency	When high overhead of supporting middle-ware is excessive When network properties are unmaskable, in practical terms