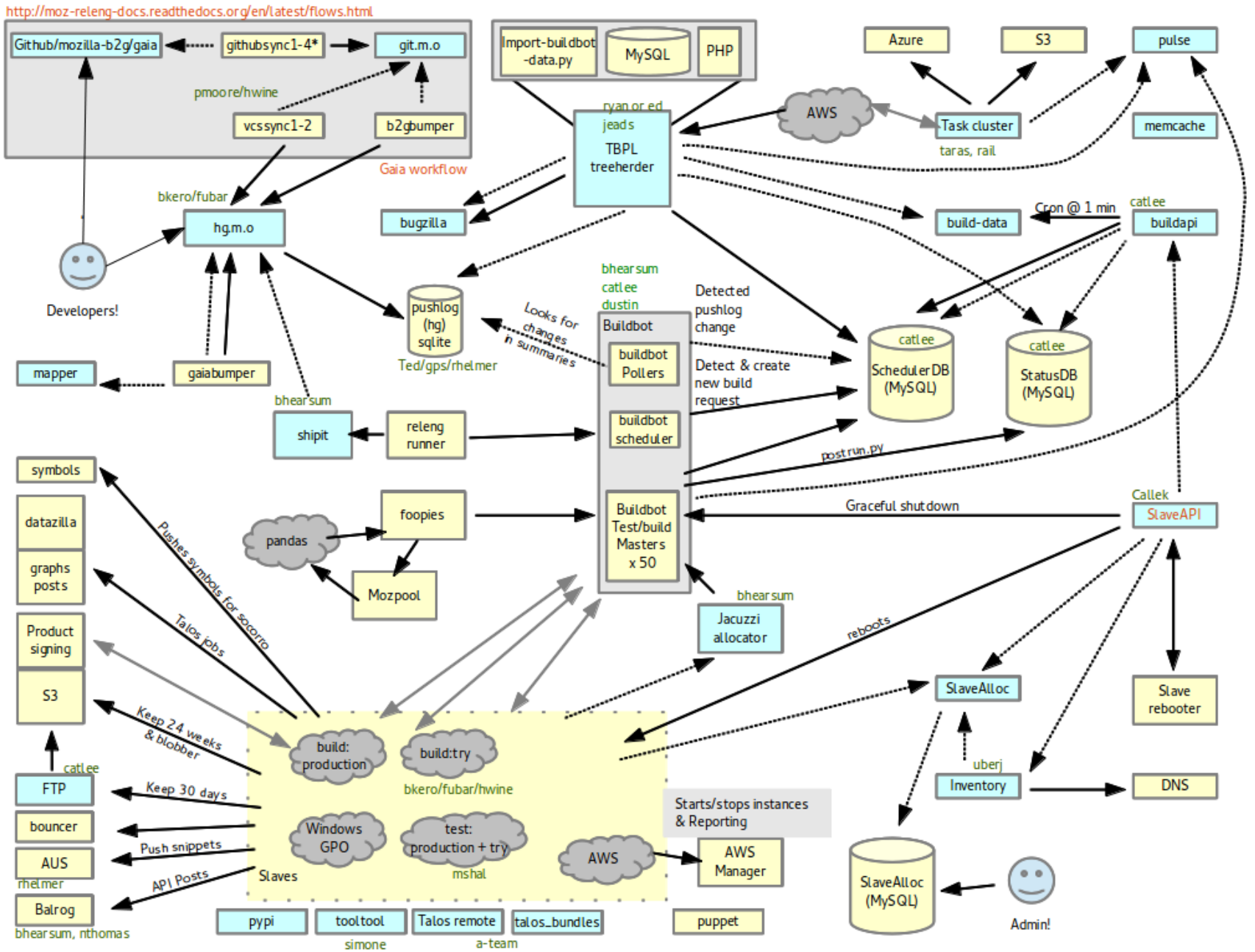
Architectural Patterns/Styles

Parinya Ekparinya

Parinya.Ekparinya@gmail.com

Software Architecture and Design

2021 Semester 1



https://www.chesnok.com/daily/2014/05/02/release-engineering-a-draft-of-an-architecture-diagram/

Structures and Views

Modern systems are frequently too complex to grasp all at once. Instead, we restrict our attention at any one moment to one (or a small number) of the software system's structures.

To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing now—which view we are taking of the architecture.

- A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them.
- A structure is the set of elements itself, as they exist in software or hardware.

Three Kinds of Structures

Architectural structures can be divided into three major categories, depending on the broad nature of the elements they show:

1. Module structures embody decisions as to how the system is to be structured as a set of code or data units that must be constructed or procured.
2. Component-and-connector structures (C&C) embody decisions as to how the system is to be structured as a set of elements that have runtime behavior (components) and interactions (connectors).
3. Allocation structures embody decisions as to how the system will relate to non-software structures in its environment (such as CPUs, file systems, networks, development teams, etc.).

Architectural pattern/style

An architectural pattern is a package of design decisions that is found repeatedly in practice. An architectural pattern establishes a relationship between:

- A context. A recurring, common situation in the world that gives rise to a problem.
- A problem. The problem, appropriately generalized, that arises in the given context. The pattern description outlines the problem and its variants and describes any complementary or opposing forces. The description of the problem often includes quality attributes that must be met.
- A solution. A successful architectural resolution to the problem, appropriately abstracted. The solution describes the architectural structures that solve the problem, including how to balance the many forces at work.

Architectural Patterns/Styles Catalog

- Layered
- Pipes and Filters
- Peer-to-Peer
- Client-Server & N-Tier
- Shared-Data (data-centric architecture)
- Model-View-Controller (MVC)
- Broker
- Publish-Subscribe
- Service-Oriented Architecture (SOA)
- Map-Reduce

We list an assortment of useful and widely used patterns. This catalog is not meant to be exhaustive—in fact no such catalog is possible!!

See: https://en.wikipedia.org/wiki/Architectural_pattern

Layered

Layered

Context:

- All complex systems experience the need to develop and evolve portions of the system independently.
- The developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem:

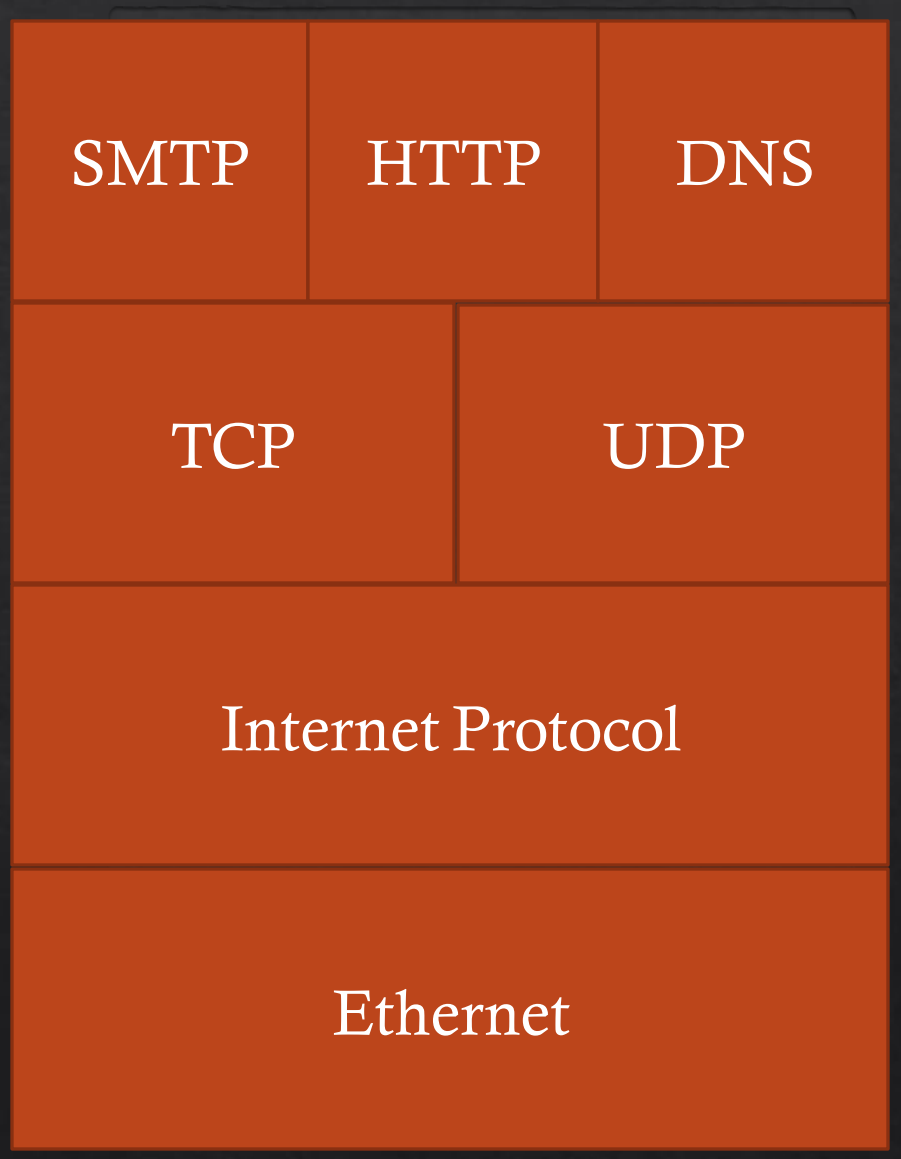
- The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

Layered Pattern Solution

Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	• Every piece of software is allocated to exactly one layer. • There are at least two layers (but usually there are three or more). • The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	• The addition of layers adds up-front cost and complexity to a system. • Layers contribute a performance penalty.

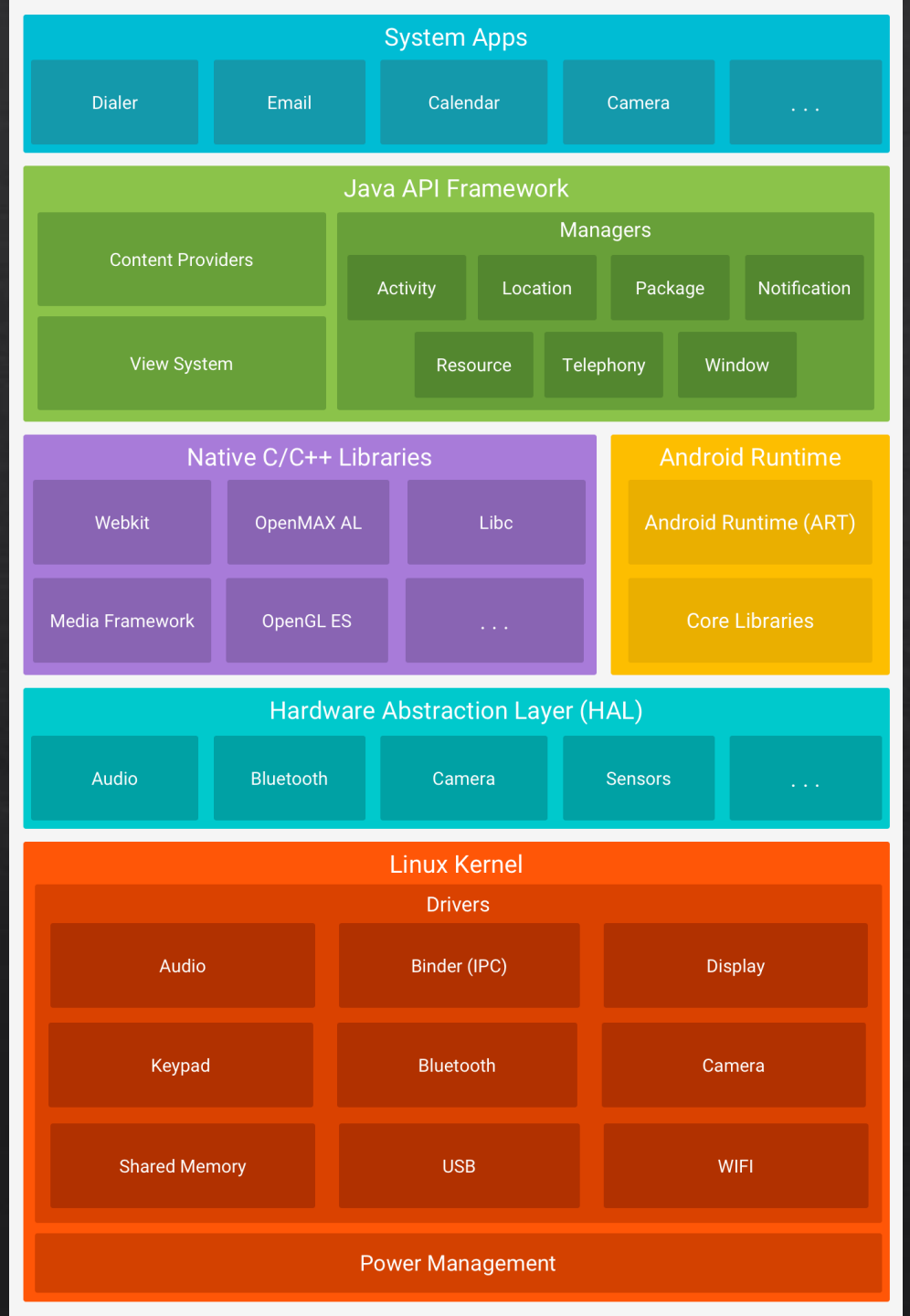
TCP/IP Protocol Suite

RFC 1122



Android Platform Architecture

https://developer.android.com/guide/platform



Layered Pattern: Resulting Qualities

แยกส่วน แก้ไขง่าย แต่เสียประสิทธิภาพ

- The layered style's constraint leads directly to the quality attributes it promotes:
  - Modifiability
  - Portability
  - Reusability
- Since a layer depends only on the layer directly beneath it, subsequent layers can be swapped or emulated.
- Taller stacks of layers yield more opportunities for substitution at the possible expense of efficient execution (performance).

Pipes and Filters

Pipes and Filters

Context:

- Many systems are required to transform streams of discrete data items, from input to output.
- Many types of transformations occur repeatedly in practice, and so it is desirable to create these as independent, reusable parts.

Problem:

- Such systems need to be divided into reusable, loosely coupled components with simple, generic interaction mechanisms. In this way they can be flexibly combined with each other.
- The components, being generic and loosely coupled, are easily reused.
- The components, being independent, can execute in parallel.

Pipe-and-Filter Pattern Solution

Elements	• *Filter,* which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. • *Pipe,* which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe preserves the sequence of data items, and it does not alter the data passing through.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	• Pipes connect filter output ports to filter input ports. • Connected filters must agree on the type of data being passed along the connecting pipe.
Weaknesses	• The pipe-and-filter pattern is typically not a good choice for an interactive system. • Having large numbers of independent filters can add substantial amounts of computational overhead. • Pipe-and-filter systems may not be appropriate for long-running computations.

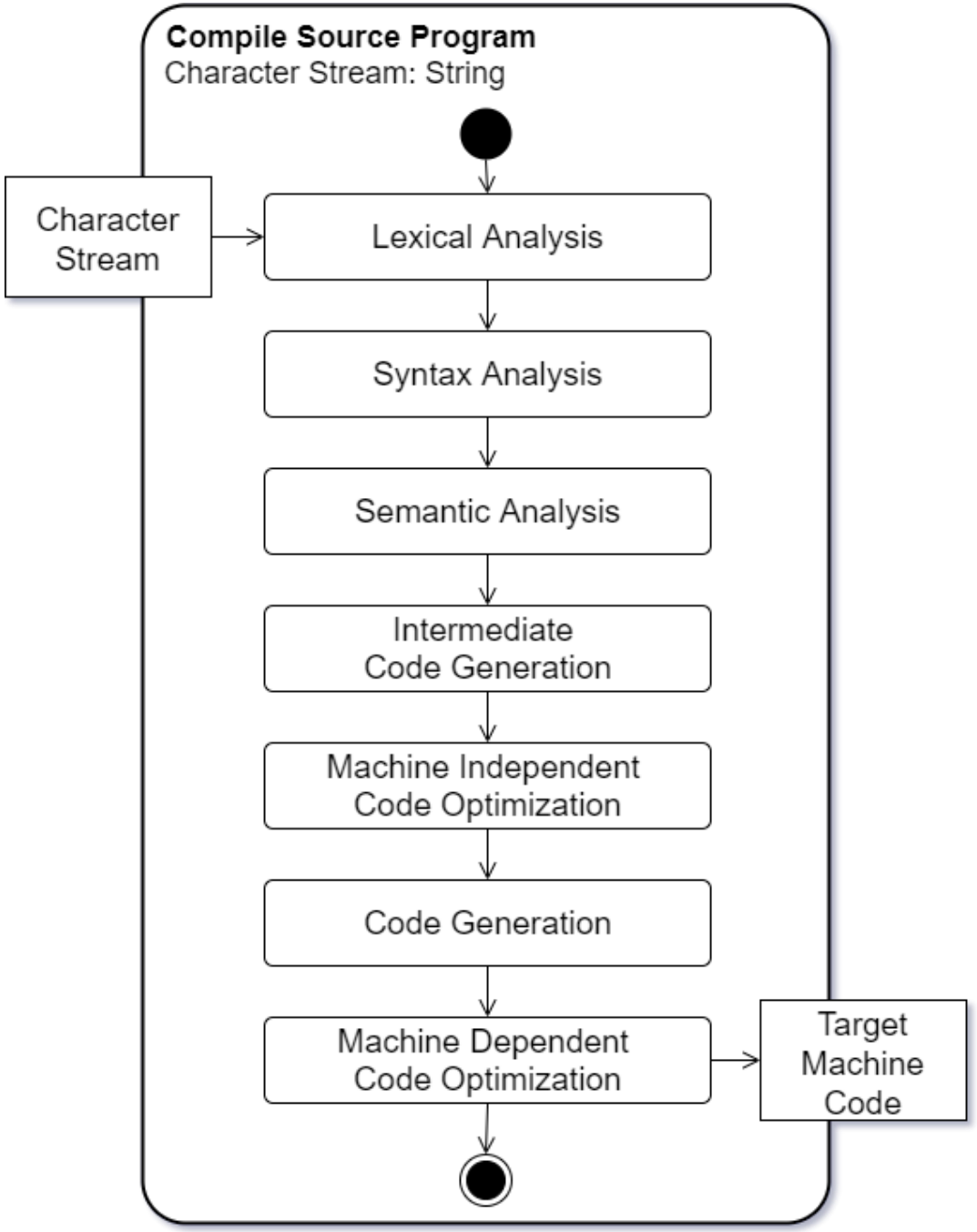
ข้อเสียคือมันควรส่งไปได้ทางเดียวเท่านั้น

Unix / Linux
Pipes and Filters

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--   1 carol doc     1605 Aug 23 07:35 macros
-rw-rw-r--   1 john  doc     2488 Aug 15 10:51 intro
-rw-rw-rw-   1 john  doc     8515 Aug  6 15:30 ch07
-rw-rw-r--   1 john  doc    14827 Aug  9 12:40 ch03
      .
      .
      .
-rw-rw-rw-   1 john  doc    16867 Aug  6 15:56 ch05
--More--(74%)
```

https://www.tutorialspoint.com/unix/unix-pipes-filters.htm

Compiler


Compile Source Program
Character Stream: String
Character Stream
Lexical Analysis
Syntax Analysis
Semantic Analysis
Intermediate Code Generation
Machine Independent Code Optimization
Code Generation
Machine Dependent Code Optimization
Target Machine Code

Pipes and Filters Pattern: Resulting Qualities

- The pipe-and-filter style promote modifiability since stages are independent of each other.
- You may not even deliver a network, just a collection of pre-made filters for others to assemble. These filters would be reused by users.
- By working within this style, opportunities for concurrency are enhanced since each filter could run in its own thread or process.
- In general, pipe-and-filter networks are inappropriate for interactive applications.

พร้อมกัน

ไม่ดีกับระบบที่ต้องคุยไปคุยกลับ

Peer-to-Peer

Peer-to-Peer

Context:

- Distributed computational entities—each of which is considered equally important in terms of initiating an interaction and each of which provides its own resources—need to cooperate and collaborate to provide a service to a distributed community of users.

Problem:

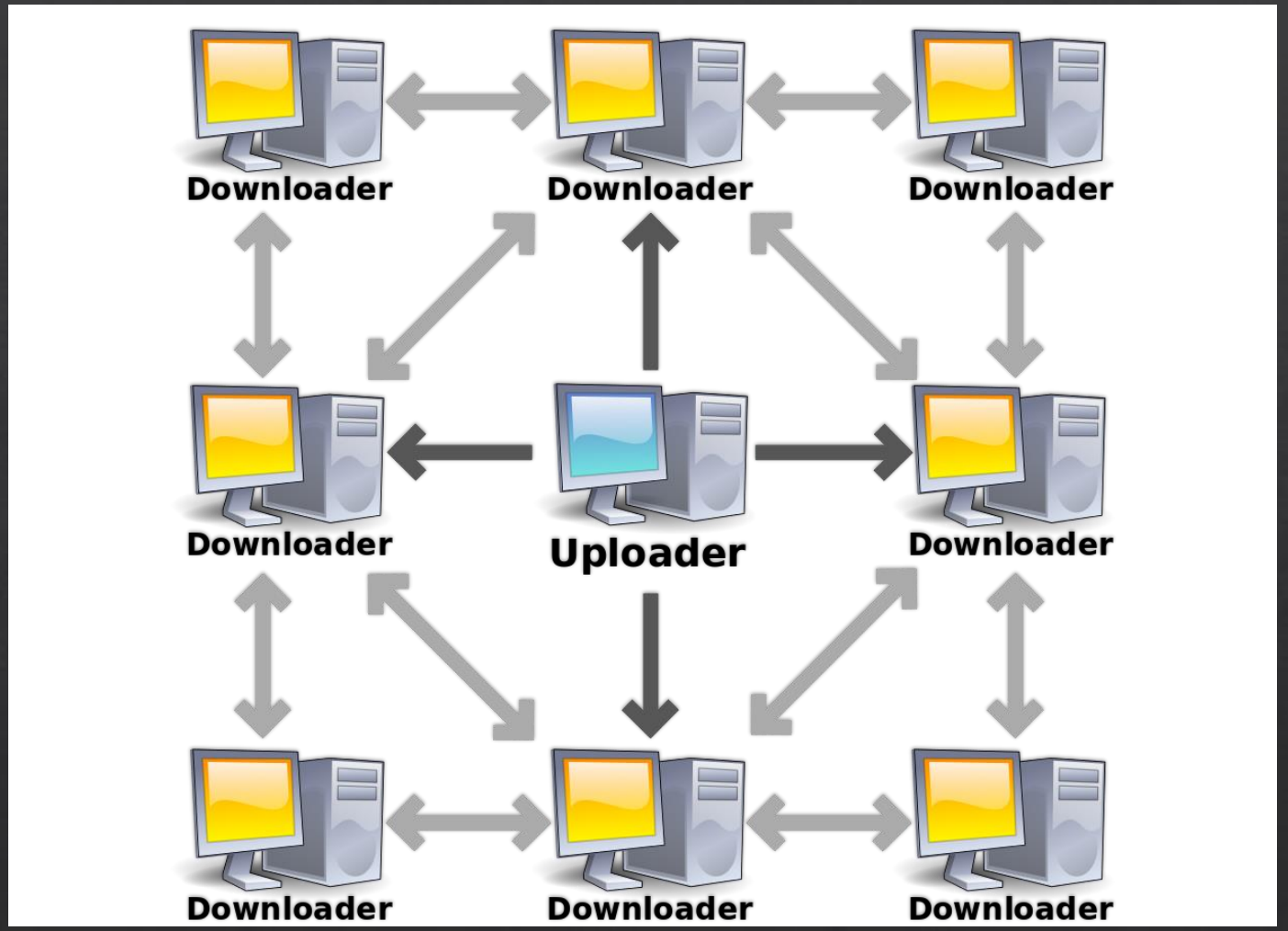
- How can a set of "equal" distributed computational entities be connected to each other via a common protocol so that they can organize and share their services with high availability and scalability?

Availability น้อย

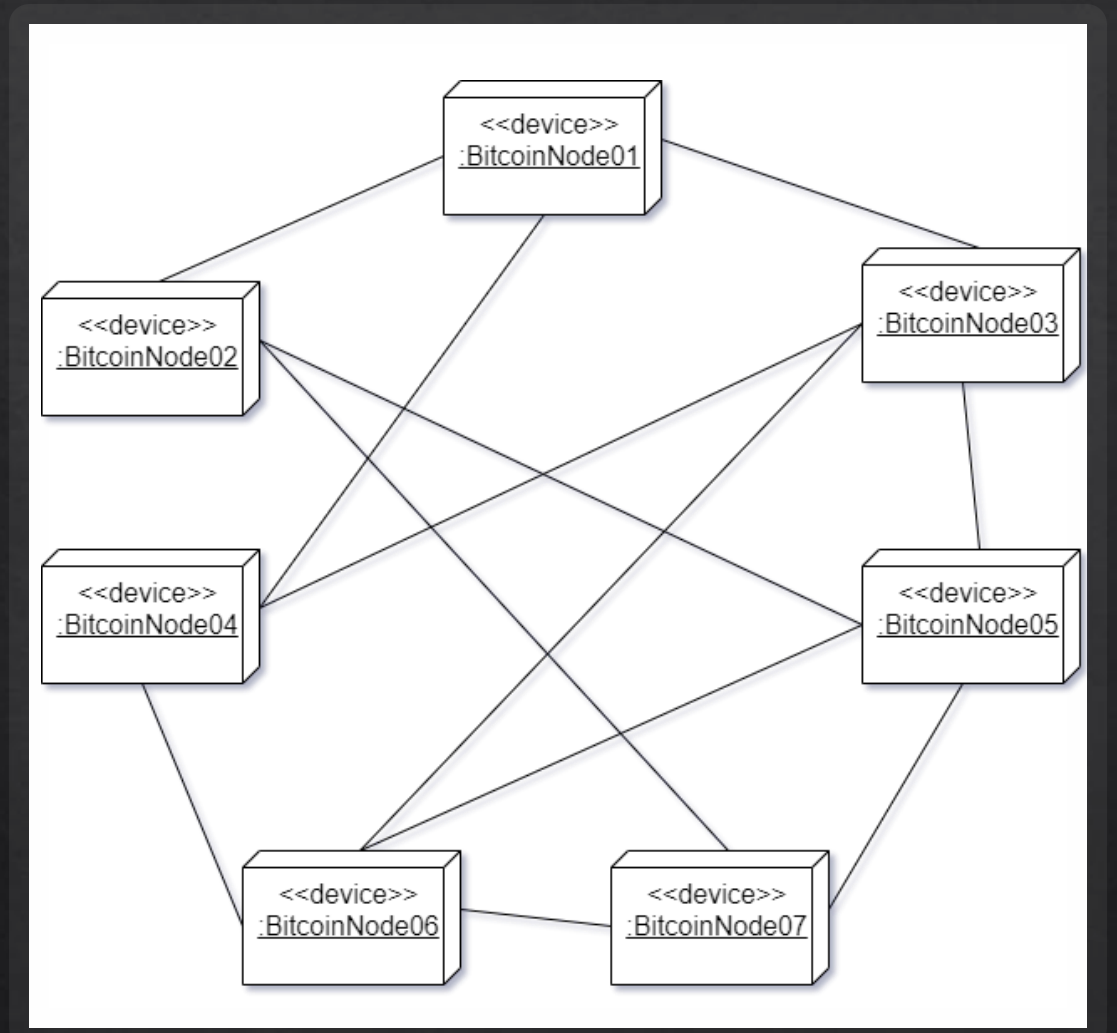
Peer-to-Peer Pattern Solution

Elements	• *Peer,* which is an independent component running on a network node. Special peer components can provide routing, indexing, and peer search capability. • *Request/reply connector,* which is used to connect to the peer network, search for other peers, and invoke services from other peers. In some cases, the need for a reply is done away with.
Relations	The relation associates peers with their connectors. Attachments may change at runtime.
Constraints	Restrictions may be placed on the following: • The number of allowable attachments to any given peer • The number of hops used for searching for a peer • Which peers know about which other peers Some P2P networks are organized with star topologies, in which peers only connect to supernodes.
Weaknesses	• Managing security, data consistency, data/service availability, backup, and recovery are all more complex. • Small peer-to-peer systems may not be able to consistently achieve quality goals such as performance and availability.

BitTorrent Network



https://en.wikipedia.org/wiki/BitTorrent#/media/File:BitTorrent_network.svg



Blockchain Network

หน้านี้ศัพท์ไม่ออก

Peer-to-Peer Pattern: Resulting Qualities

สุง

- It also promotes availability, since failures of individual nodes are less likely to impair the system. แต่การรันตรียากเฉย ๆ
- Peer-to-peer network has no single point of failure, and no central infrastructure is needed. The network has high scalability.
- The lack of a centralized control makes it difficult to enforce security in peer-to-peer pattern.

Client-Server & N-Tier

Client-Server & N-Tier

Context:

- There are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control access or quality of service.

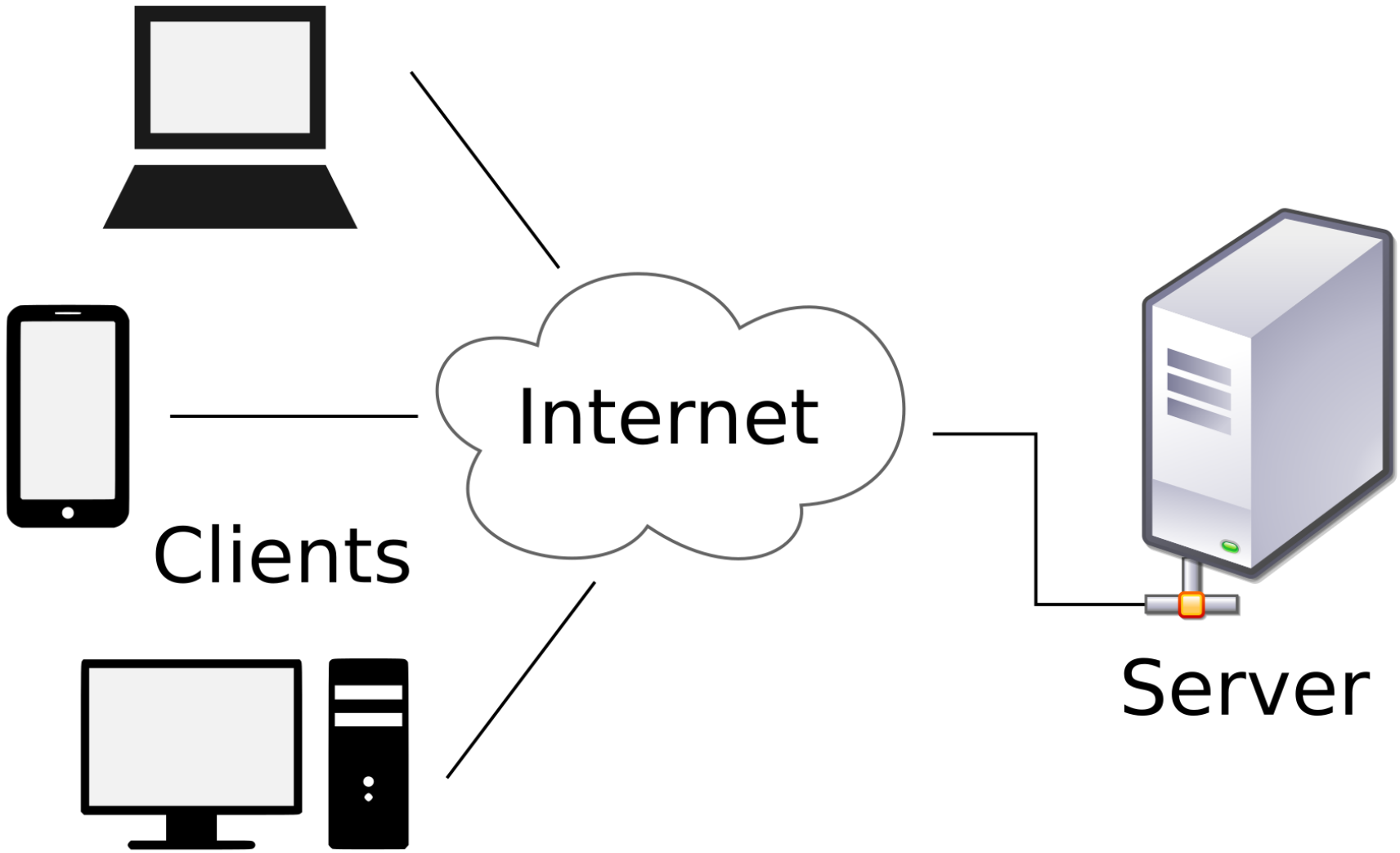
Problem:

- By managing a set of shared resources and services, we can promote modifiability and reuse, by factoring out common services and having to modify these in a single location, or a small number of locations.
- We want to improve scalability and availability by centralizing the control of these resources and services, while distributing the resources themselves across multiple physical servers.

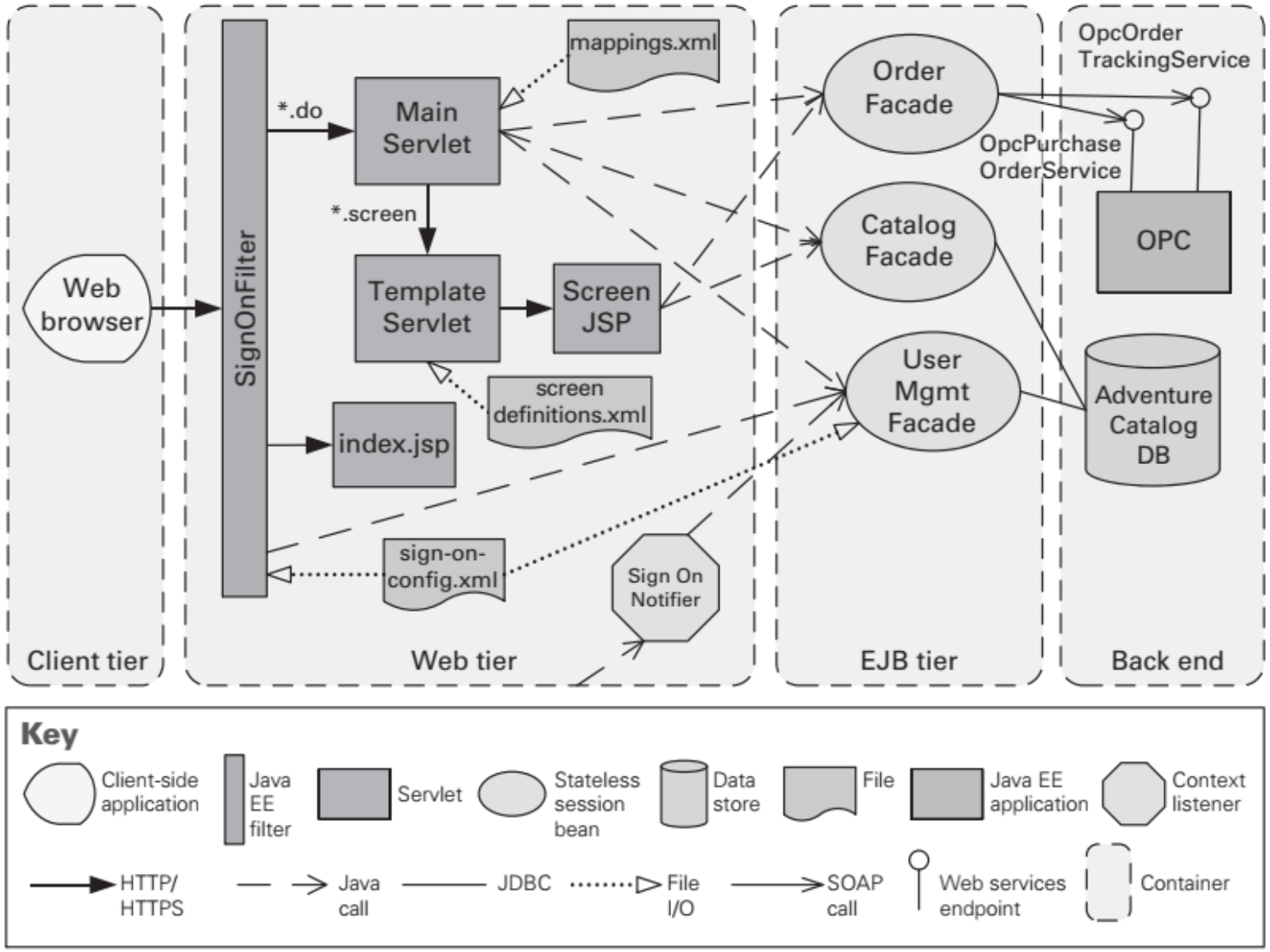
Chain กันไปเรื่อย ๆ

Client-Server Pattern Solution

Elements	• *Client,* a component that invokes services of a server component. Clients have ports that describe the services they require. • *Server,* a component that provides services to clients. Servers have ports that describe the services they provide. • *Request/reply connector,* a data connector employing a request/reply protocol, used by a client to invoke services on a server.
Relations	The *attachment* relation associates clients with servers.
Constraints	• Clients are connected to servers through request/reply connectors. • Server components can be clients to other servers. • Components may be arranged in tiers, which are logical groupings of related functionality or functionality that will share a host computing environment
Weaknesses	• Server can be a performance bottleneck. • Server can be a single point of failure. • Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built.



https://en.wikipedia.org/wiki/Client%E2%80%93server_model#/media/File:Client-server-model.svg



Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Client-Server Pattern: Resulting Qualities

- The client-server style establishes an asymmetrical power relationship between the client and server in regards to who can initiate processing. The server often ends up with more influence since it is providing the service.
- An organization can change a business process or rule by changing its implementation in one place, the server, rather than across the many clients, so maintainability is enhanced.

Shared-Data
(data-centric architecture)

Shared-Data

Context:

- Various computational components need to share and manipulate large amounts of data.
- This data does not belong solely to any one of those components.

Problem:

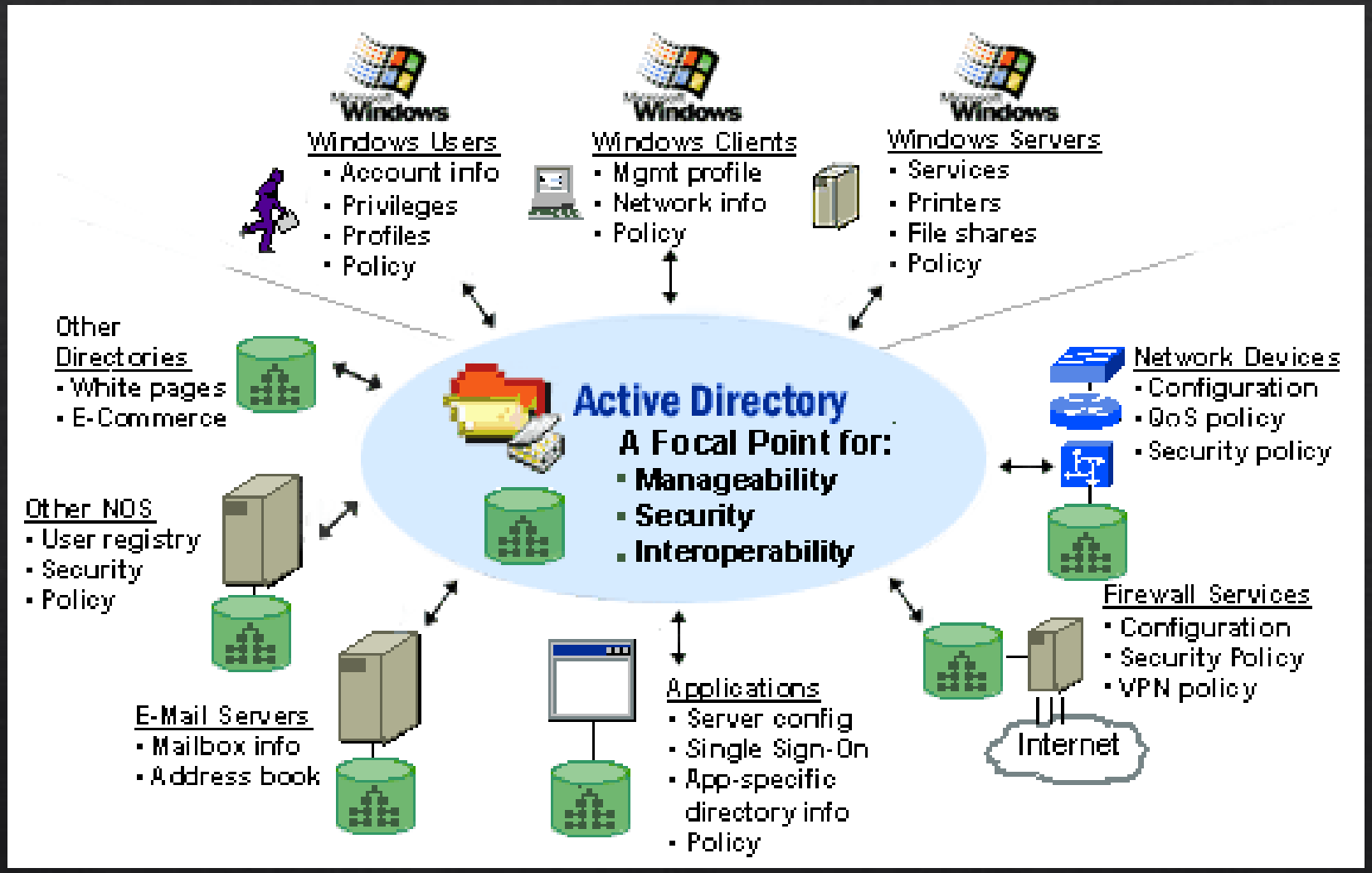
- How can systems store and manipulate persistent data that is accessed by multiple independent components?

คุยกันโดยเขียนลงตรงกลางแล้วคนอื่นไปอ่านเอา

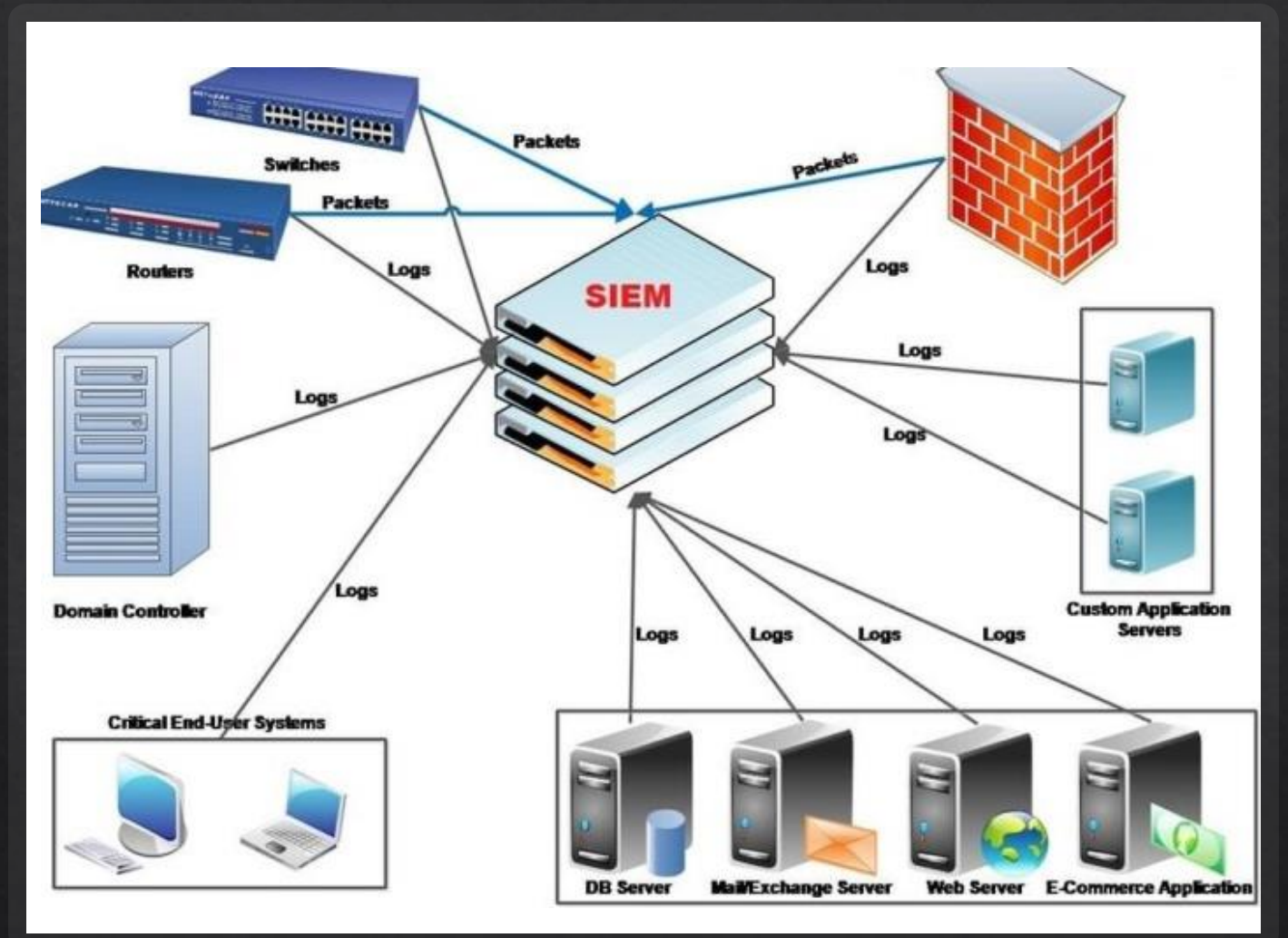
Shared-Data Pattern Solution

Elements	• *Shared-data store.* Concerns include types of data stored, data performance-oriented properties, data distribution, and number of accessors permitted. • *Data accessor component*. • *Data reading and writing connector*.
Relations	*Attachment* relation determines which data accessors are connected to which data stores.
Constraints	Data accessors interact with the data store(s).
Weaknesses	• The shared-data store may be a performance bottleneck. • The shared-data store may be a single point of failure. • Producers and consumers of data may be tightly coupled.

Active Directory



http://ww2.cs.fsu.edu/~bogdanov/SysAdminSp05/Agenda/week07/ad1.htm



Security Information and Event Management

https://gbhackers.com/security-information-and-event-management-siem-a-detailed-explanation/

Shared-Data Pattern: Resulting Qualities

- Use of this pattern has the effect of decoupling the producer of the data from the consumers of the data; hence, this pattern supports modifiability, as the producers do not have direct knowledge of the consumers.
- There are several potential problems associated with this pattern:
  - For one, the shared-data store may be a performance bottleneck.
  - The shared-data store is also potentially a single point of failure.

Model-View-Controller (MVC)

Model-View-Controller (MVC)

นิยมออกแบบเว็ปแอฟ

Context:

- User interface software is typically the most frequently modified portion of an interactive application. For this reason, it is important to keep modifications to the user interface software separate from the rest of the system.
- Users often wish to look at data from different perspectives, such as a bar graph or a pie chart. These representations should both reflect the current state of the data.

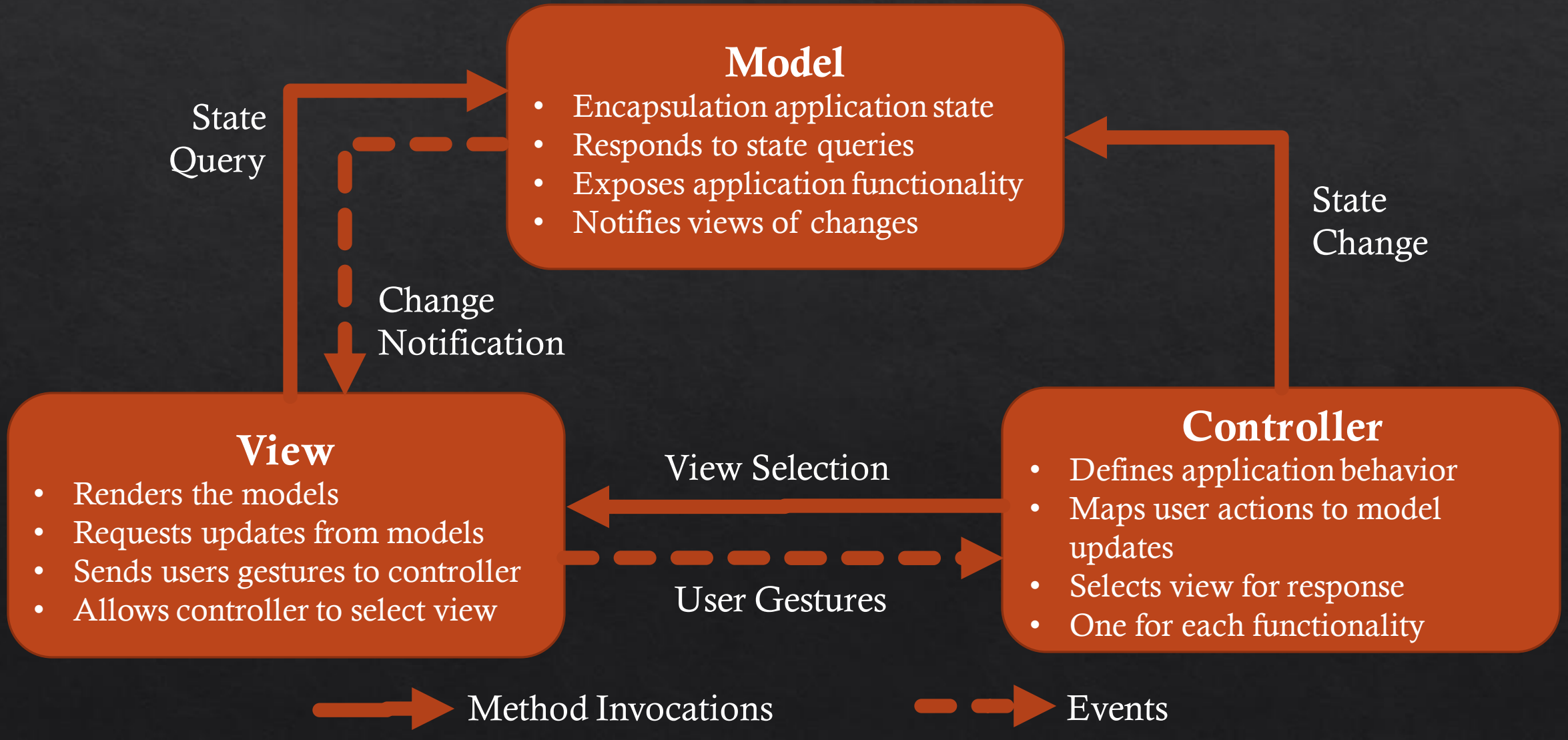
Problem:

- How can user interface functionality be kept separate from application functionality and yet still be responsive to user input, or to changes in the underlying application's data?
- And how can multiple views of the user interface be created, maintained, and coordinated when the underlying application data changes?

MVC Pattern Solution

Elements	• The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic. • The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both. • The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view.
Relations	The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes.
Constraints	• There must be at least one instance each of model, view, and controller. • The model component should not interact directly with the controller.
Weaknesses	• The complexity may not be worth it for simple user interfaces. • The model, view, and controller abstractions may not be good fits for some user interface toolkits.

A Common MVC Implementation



MVC Pattern: Resulting Qualities

- Modifiability is enhanced because of the independence of view and controller components from the model component and the minimal dependencies.
- The system is extensible since unanticipated views and controllers are easy to add later.
- It can be easier to manage and persist state since it is centralized in the model component.

Broker

Broker

Context:

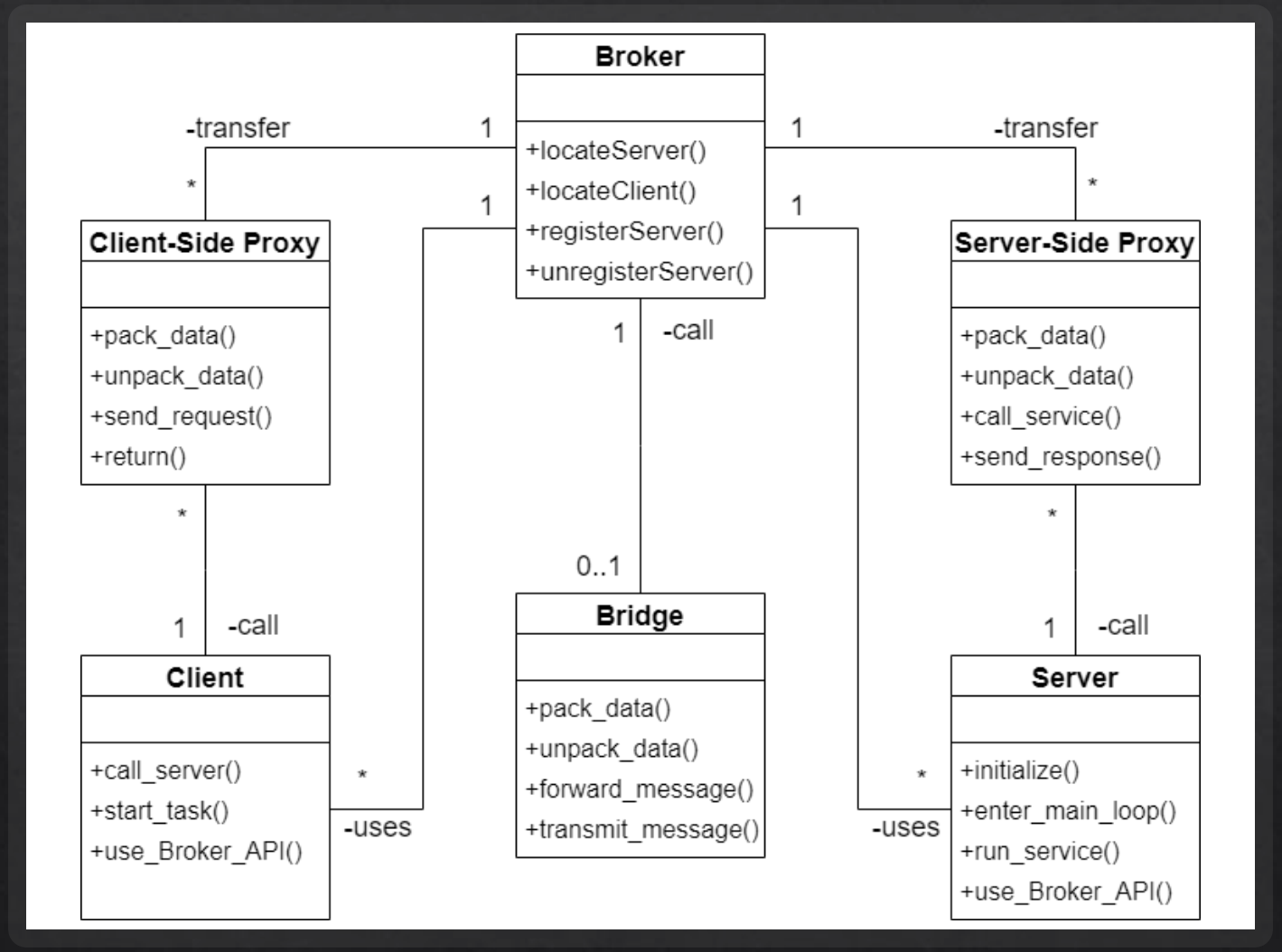
- Many systems are constructed from a collection of services distributed across multiple servers.
- Implementing these systems is complex because you need to worry about how the systems will interoperate—how they will connect to each other and how they will exchange information—as well as the availability of the component services.

Problem:

- How do we structure distributed software so that service users do not need to know the nature and location of service providers, making it easy to dynamically change the bindings between users and providers?

Broker Pattern Solution

Elements	• *Client,* a requester of services • *Server,* a provider of services • *Broker,* an intermediary that locates an appropriate server to fulfill a client's request, forwards the request to the server, and returns the results to the client • *Client-side / Server-side proxy,* an intermediary that manages the actual communication with the broker, including marshaling, sending, and unmarshaling of messages
Relations	The *attachment* relation associates clients (and, optionally, client-side proxies) and servers (and, optionally, server-side proxies) with brokers.
Constraints	The client can only attach to a broker (potentially via a client-side proxy). The server can only attach to a broker (potentially via a serverside proxy).
Weaknesses	• Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck. • The broker can be a single point of failure. • A broker adds up-front complexity. • A broker may be a target for security attacks. • A broker may be difficult to test.



Class Diagram for Broker Pattern

Broker Pattern: Resulting Qualities

The broker pattern provides all of:

- the modifiability benefits of the use-an-intermediary tactic
- an availability benefit (because the broker pattern makes it easy to replace a failed server with another)
- a performance benefit (because the broker pattern makes it easy to assign work to the least-busy server)
- an interoperability benefit (because the broker pattern provides mechanisms to discover and locate services)

However, the pattern also carries with it some liabilities. The use of this pattern adds the overhead of the intermediary and thus latency.

Publish-Subscribe

Publish-Subscribe

Context:

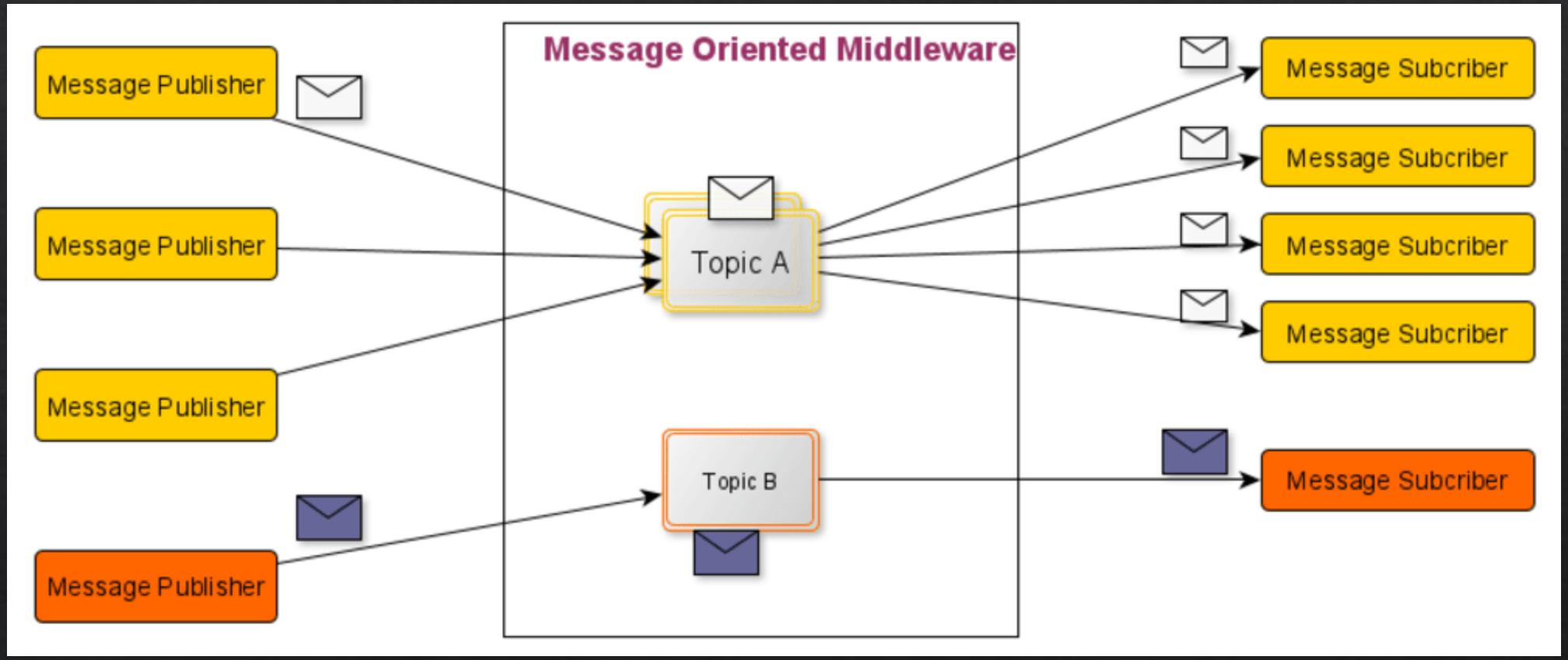
- There are several independent producers and consumers of data that must interact.
- The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

Problem:

- How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

Publish-Subscribe Pattern Solution

Elements	*Any C&C component* with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events. *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
Relations	The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.
Constraints	All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system. A component may be both a publisher and a subscriber, by having ports of both types.
Weaknesses	• Typically increases latency and has a negative effect on scalability and predictability of message delivery time. • Less control over ordering of messages, and delivery of messages is not guaranteed.



https://dev.to/tranthanhdeveloper/point-to-point-and-publish-subscribe-messaging-model-41j0

Publish-Subscribe Pattern: Resulting Qualities

The publish-subscribe pattern decouples producers and consumers of events. Therefore, it improves modifiability.

- When a new component needs to do work based on an event, it can simply subscribe to that event and the event publisher is unchanged.
- A new event publisher can be added without affecting the system, and later a component (new or existing) can begin subscribing to those events.

The event bus adds a layer of indirection between producers and consumers. It can therefore hurt the system's performance.

Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA)

Context:

- Several services are offered (and described) by service providers and consumed by service consumers.
- Service consumers need to be able to understand and use these services without any detailed knowledge of their implementation.

Problem:

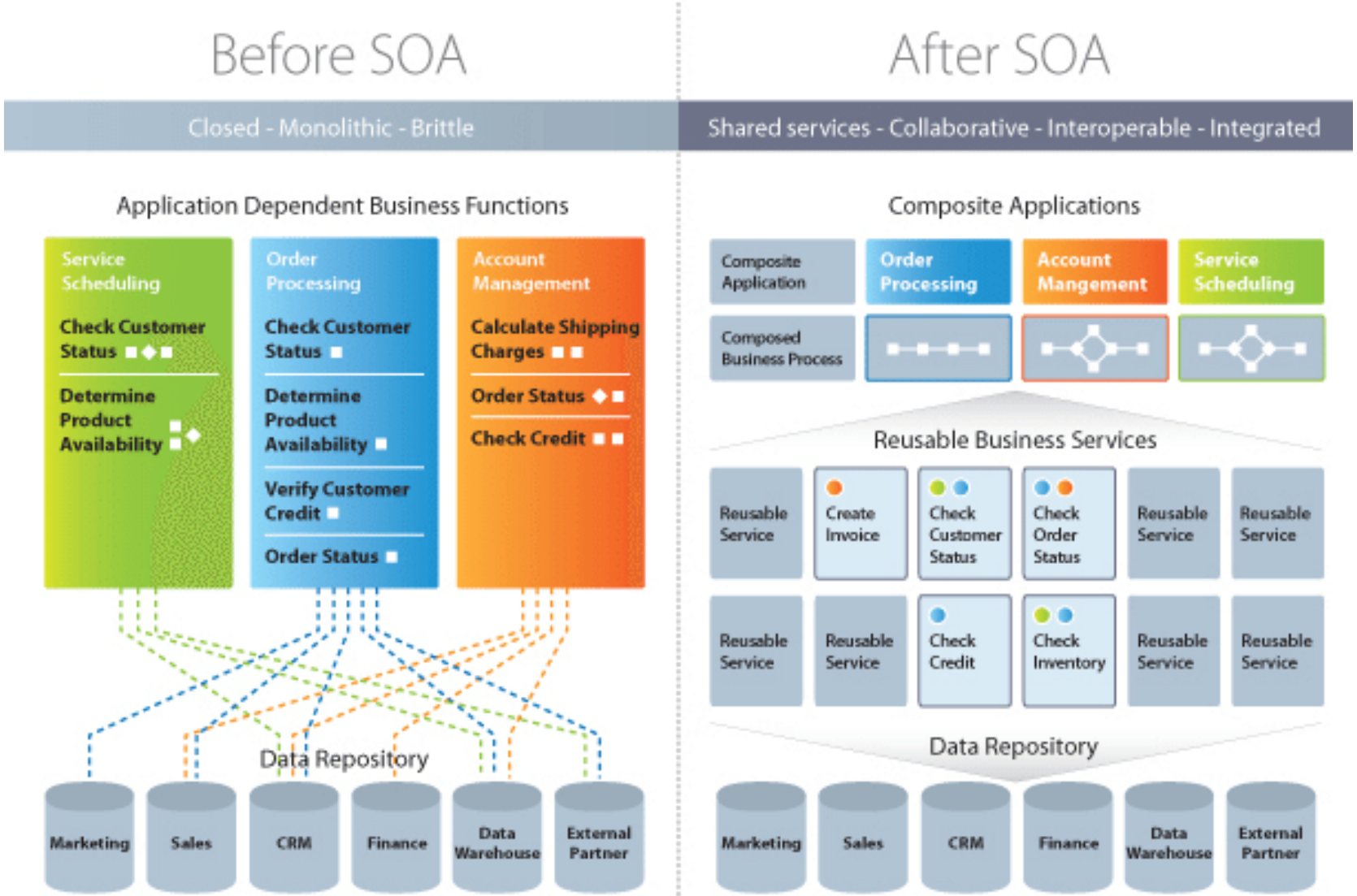
- How can we support interoperability of distributed components running on different platforms and written in different implementation languages, provided by different organizations, and distributed across the Internet?
- How can we locate services and combine (and dynamically recombine) them into meaningful coalitions while achieving reasonable performance, security, and availability?

SOA Pattern Solution (1)

Elements	• *Service providers,* which provide one or more services through published interfaces. Concerns are often tied to the chosen implementation technology, and include performance, authorization constraints, availability, and cost. In some cases these properties are specified in a service-level agreement. • *Service consumers,* which invoke services directly or through an intermediary. *Service providers* may also be service consumers. • *ESB,* which is an intermediary element that can route and transform messages between service providers and consumers. • *Registry of services,* which may be used by providers to register their services and by consumers to discover services at runtime. • *Orchestration server,* which coordinates the interactions between service consumers and providers based on languages for business processes and workflows. • *Connector,* which is used to connect between components. In practice, SOA environments may involve a mix of various connectors along with legacy protocols and other communication alternatives.

SOA Pattern Solution (2)

Relations	Attachment of the different kinds of components available to the respective connectors, for example: • *SOAP connector,* which uses the SOAP protocol for synchronous communication between web services, typically over HTTP. • *REST connector,* which relies on the basic request/reply operations of the HTTP protocol. • *Asynchronous messaging connector,* which uses a messaging system to offer point-to-point or publish-subscribe asynchronous message exchanges.
Constraints	Service consumers are connected to service providers, but intermediary components (e.g., ESB, registry, orchestration server) may be used.
Weaknesses	• SOA-based systems are typically complex to build. • You don't control the evolution of independent services. • There is a performance overhead associated with the middleware, and services may be performance bottlenecks, and typically do not provide performance guarantees.



https://medium.com/sirisoft/eai-soa-esb-microservices-35ca49503d12

SOA Pattern: Resulting Qualities

- The main benefit and the major driver of SOA is interoperability. SOA is often used to integrate a variety of systems, including legacy systems.
- Another benefit of SOA is modifiability. Special SOA components such as the registry or the ESB also allow dynamic reconfiguration, which is useful when there's a need to replace or add versions of components with no system interruption.
- The SOA pattern can be quite complex to design and implement.
- Other potential problems with this pattern include the performance overhead of the middleware that is interposed between services and clients,

Microservice architecture

- There is no formal definition of the microservices architectural style. A consensus view has evolved over time in the industry.
- Some agree that microservice architecture is a variant of SOA.
  - Recommended reading: https://martinfowler.com/articles/microservices.html
- Some point out the differences to distinguish between SOA and microservice architecture.
  - Recommended reading: https://www.ibm.com/cloud/blog/soa-vs-microservices
- One important aspect of microservice architecture is the independence of components. It minimizes the need to share components among its applications to promote a high degree of loose coupling.

Map-Reduce

Map-Reduce

Context:

- Businesses have a pressing need to quickly analyze enormous volumes of data they generate or access, at petabyte scale. Examples include logs of interactions in a social network site, massive document or data repositories, and pairs of <source, target> web links for a search engine.
- Programs for the analysis of this data should be easy to write, run efficiently, and be resilient with respect to hardware failure.

Problem:

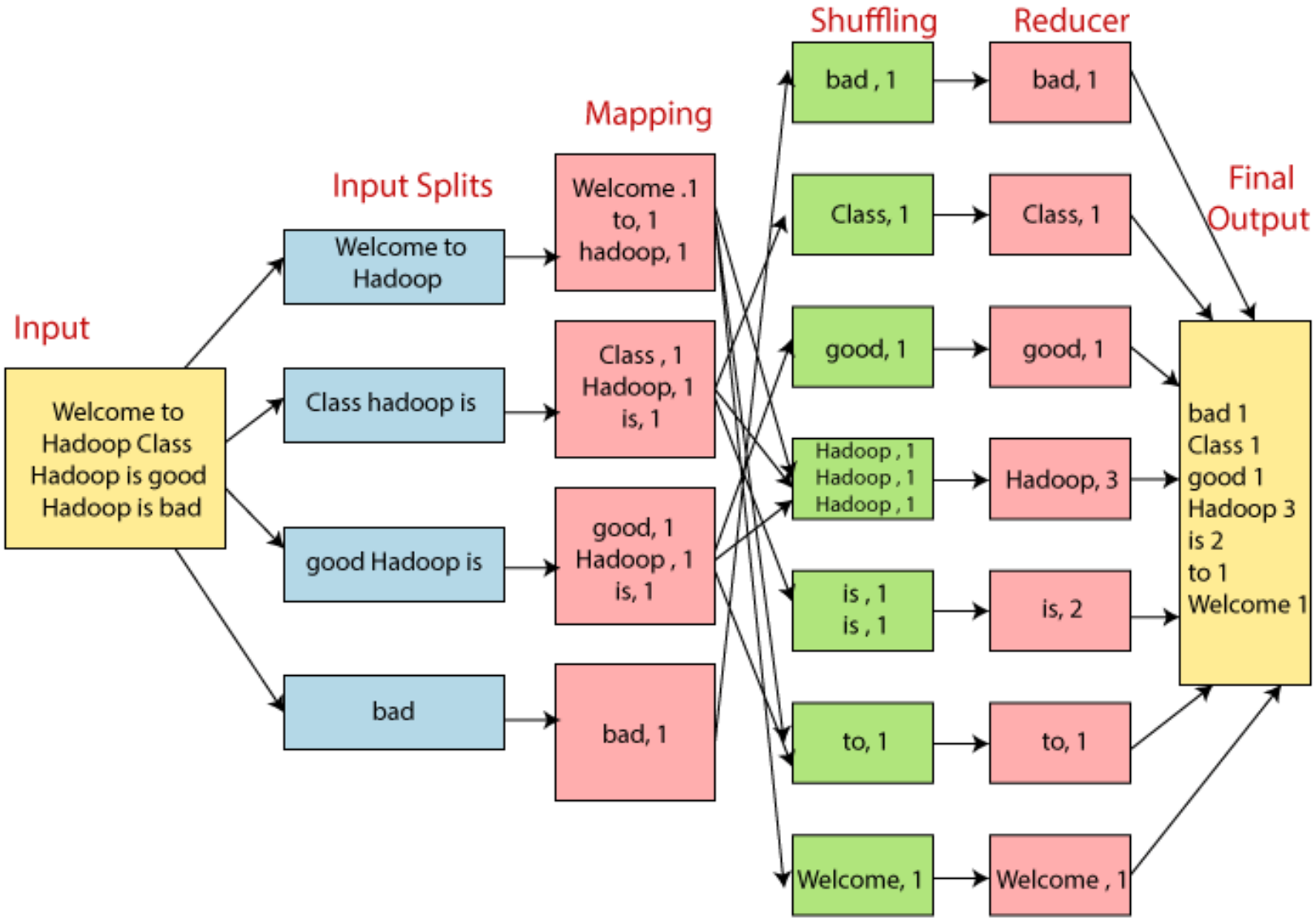
- How to efficiently perform a distributed and parallel sort of a large data set and provide a simple means for the programmer to specify the analysis to be done?

Map-Reduce Pattern Solution (1)

Elements	• *Map* is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis. • *Reduce* is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load. • The *infrastructure* is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure.
Relations	*Deploy on* is the relation between an instance of a map or reduce function and the processor onto which it is installed. *Instantiate, monitor, and control* is the relation between the infrastructure and the instances of map and reduce.

Map-Reduce Pattern Solution (2)

Constraints	• The data to be analyzed must exist as a set of files. • The map functions are stateless and do not communicate with each other. • The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs.
Weaknesses	• If you do not have large data sets, the overhead of map-reduce is not justified. • If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost. • Operations that require multiple reduces are complex to orchestrate.



https://www.tutorialandexample.com/mapreduce-tutorial/

Map-Reduce Pattern: Resulting Qualities

- The primary quality attribute that map-reduce improves is scalability. Tasks that were impractical to compute with a single computer can be divided across many machines, improving the performance.
- Note that the performance of this pattern is heavily influenced by data locality. Intermediate results need to be kept close to the map and reduce worker components to avoid network bandwidth use.
- Once a program is written to use the map-reduce style, it could run on a cluster of one, or one thousand, machines. Map-reduce also promotes availability, since it recovers from machine failures by rescheduling the work on another machine.

Summary

- Layered
- Pipes and Filters
- Peer-to-Peer
- Client-Server & N-Tier
- Shared-Data (data-centric architecture)
- Model-View-Controller (MVC)
- Broker
- Publish-Subscribe
- Service-Oriented Architecture (SOA)
- Map-Reduce