EECE 310

Software Design: Architecture

Learning Goals

- Describe the context (goals and constraints) of software design
- Understand the essence of **software architecture** and why it is needed
- Describe how **architectural styles** can be formed
- Distinguish between different architectural styles

What is Software Design?

What is design?

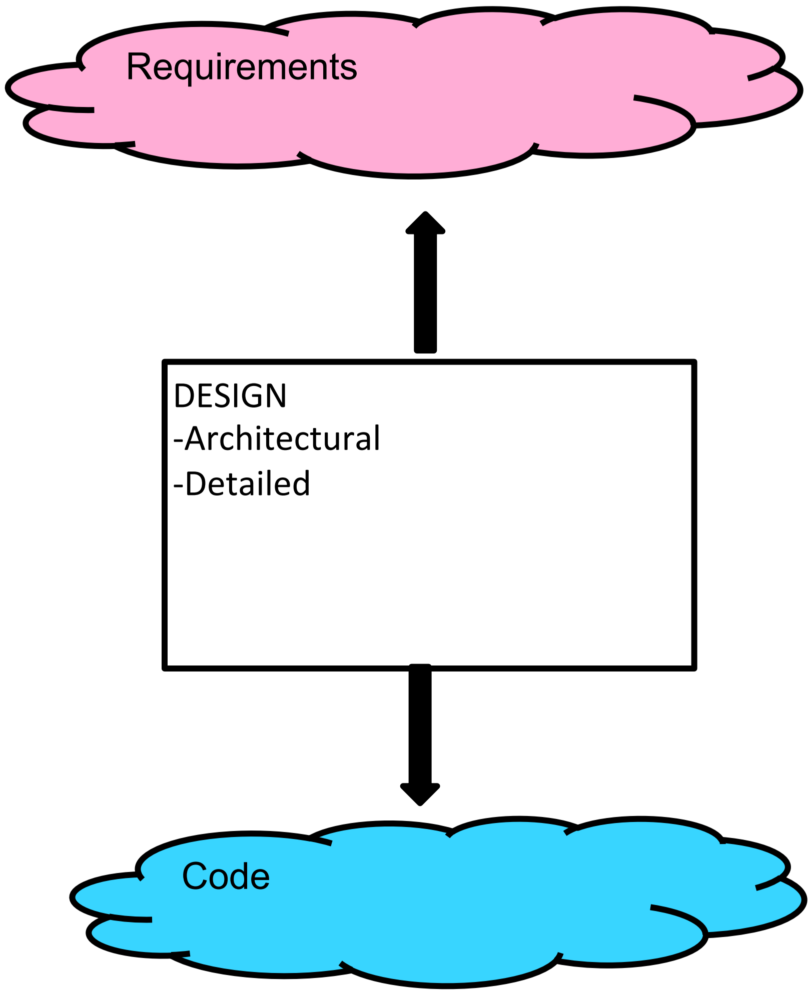
What makes something a design problem? It’s where you stand with ***a foot in two worlds*** *– the world of technology and the world of people and human purposes – and you try to bring the two together.*

- Mitch Kapor, A Software Design Manifesto (1991)

Software Design(er)

- Software design is **NOT** user interface design
- Software designers
  - are concerned with the overall product conception
  - should have strong technical grounding
  - work in conjunction with developers

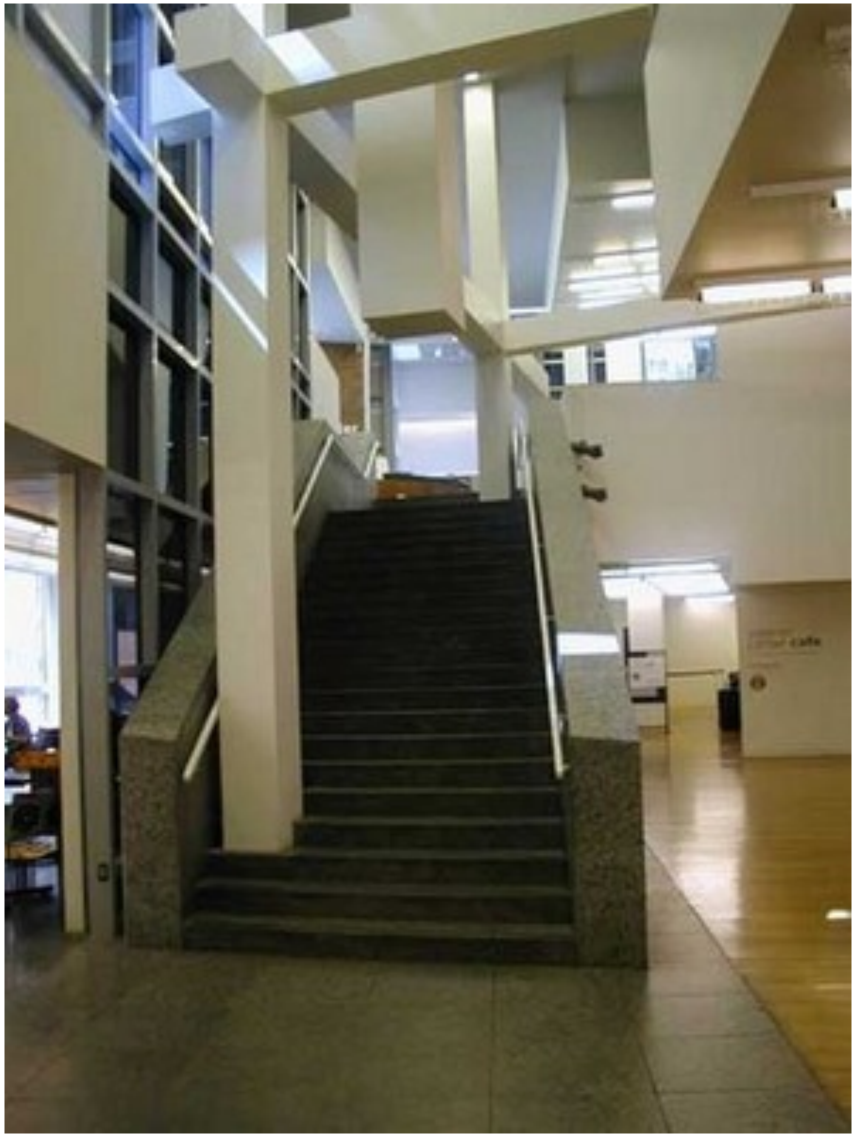
Design to Bridge the Gap



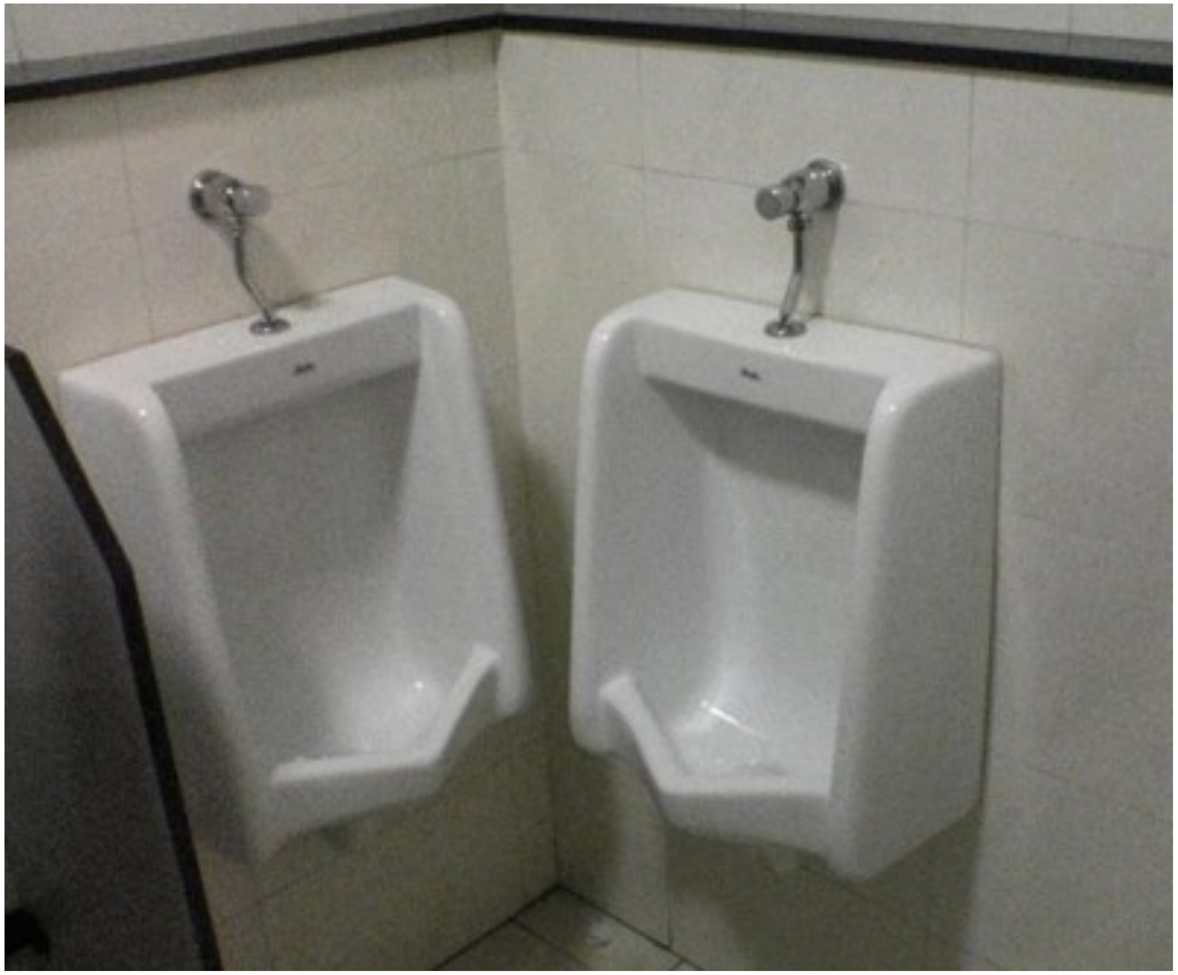
Why Design?

- Facilitates communication
- Eases system understanding
- Eases implementation
- Helps discover problems early
- Increases product quality
- Reduces maintenance costs
- Facilitates product upgrade

Cost of not thinking ahead…



Bad design (and impl.)



How to approach Design?

"Treat design as a wicked, sloppy, ***heuristic*** process. Don't settle for the first design that occurs to you. ***Collaborate.*** Strive for ***simplicity***. Prototype when you need to. Iterate, iterate and ***iterate again***. You'll be happy with your designs."

Steve McConnell. *Code Complete*. Ch. 5

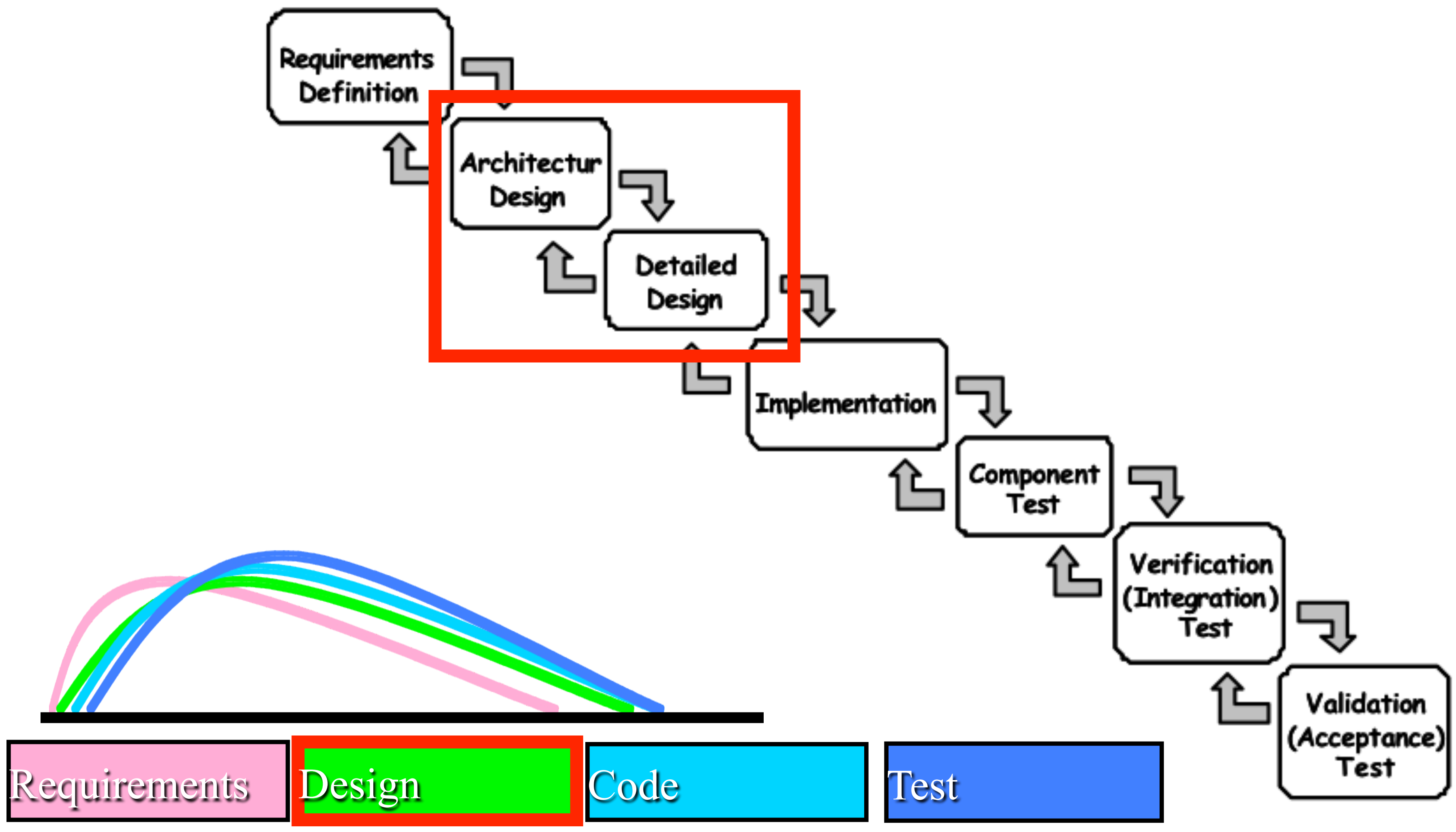
How to Design?

- Start global
- Subdivide
- Iterate

Two common phases of Software Design

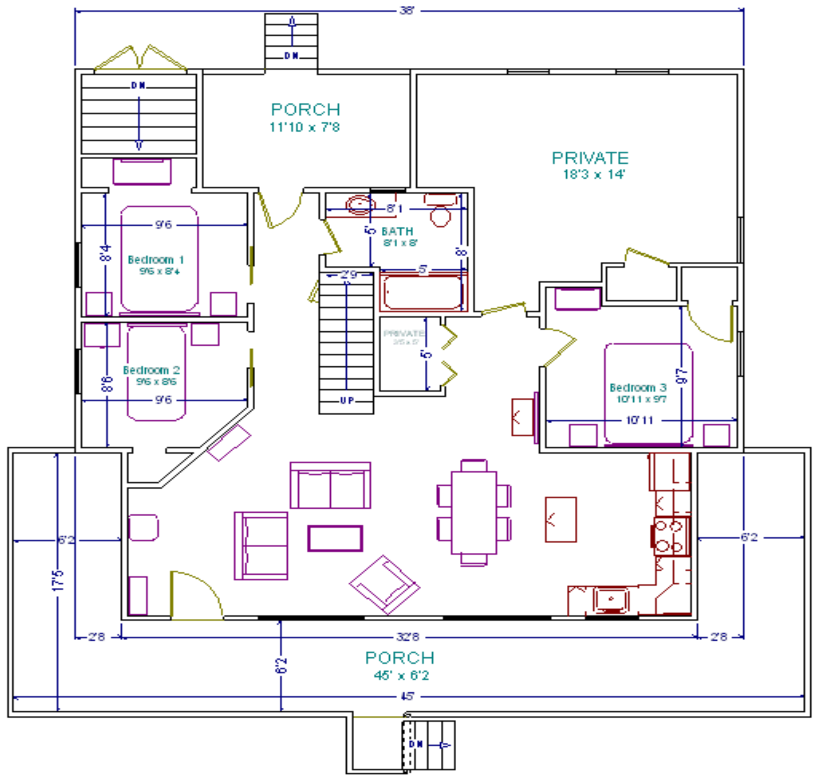
1. **Architectural** design
   - Determining which sub-systems you need (e.g., web server, DB…)
   - Discussed today
2. **Detailed** design
   - Next lecture

Where does it fit in process?



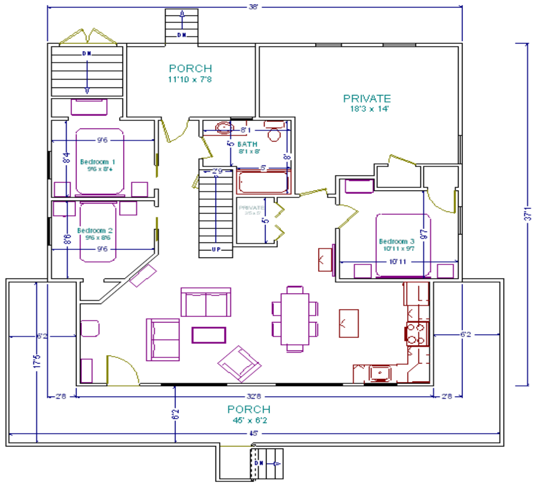
Architectural Design

- Denotes the high-level structure of a software system
- Main components and their behavior
  - Connections between the components / communication
- Architectural Styles: reusable architectural designs
  - data-flow, client/server, REST, …
- Tools: UML, whiteboard, paper



38'
DN
PORCH
11'10 x 7'8
PRIVATE
18'3 x 14'
9'6
8'4
Bedroom 1
9'6 x 8'4
8'1
5'
BATH
8'1 x 8'
8'
2'9
5'
PRIVATE
5'
Bedroom 2
9'6 x 8'6
8'6
9'6
UP
9'7
Bedroom 3
10'11 x 9'7
10'11
37'1
6'2
17'5
6'2
2'8
32'8
2'8
6'2
PORCH
45' x 6'2
45'
DN

Outline



- Distributed and networked architectures
  - Limitations
  - Common styles
- Architectural Styles
  - Definition, examples
- Decentralized applications
  - Peer-to-peer
  - Web services
- Web-based applications
  - Classical web architecture: REST
  - Modern Web 2.0 applications

Network Application Architecture

- Software architecture of a network-based app
  - abstract system view and model for comparison
  - communication restricted to message passing
- Defines
  - How **components** are allocated and identified
  - How components **interact** to form a system
  - The amount and granularity of **communication** needed for interaction
  - Interface **protocols**

Ideal Properties of Distributed Systems

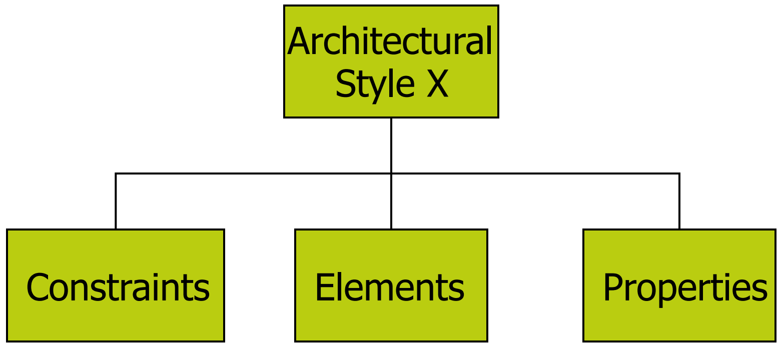
- The network is reliable
- Latency is zero
- Bandwidth is infinite
- The network is secure
- Topology doesn’t change
- There is one administrator
- The network is homogeneous

-- Deutsch & Gosling

The best network application performance is obtained by not using the network!

Architectural Style: Definition

"A set of architectural constraints restricting the roles of architectural elements, inducing the architectural properties desired of a system, when given a name, becomes an *architectural style*."

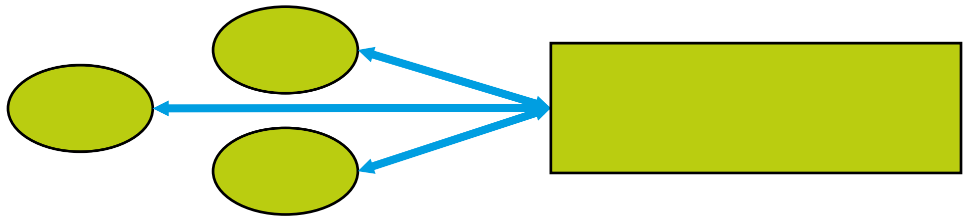


Architectural Styles

- **Common patterns** within system architectures
- One system may be composed of multiple styles
- Some styles are hybrids of other styles
- An architecture is an instantiation of a style
- We could equally talk about
  - computer architecture
  - network architecture
  - software architecture
    - network-based application architecture

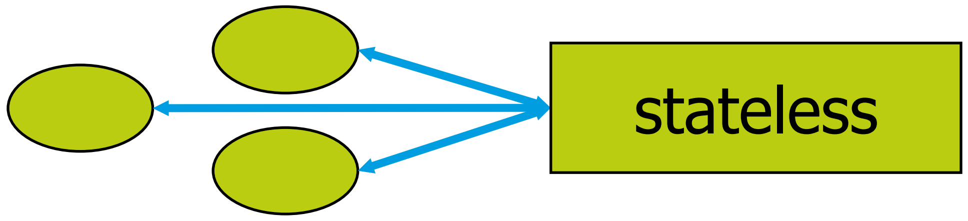
Architectural Style: Examples?

Client/Server (CS)



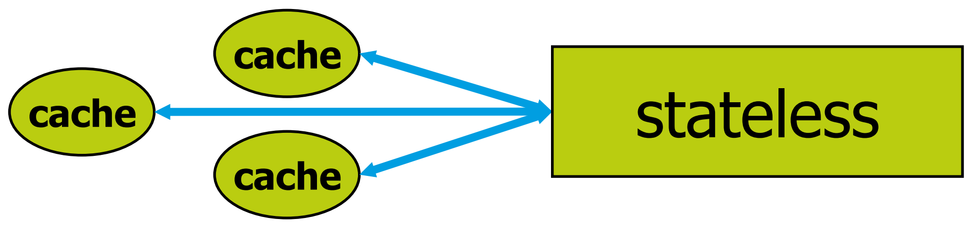
- most well-known, least meaningful style
- initiators (clients) and listeners (servers)
- emphasizes separation of concerns

Client-Stateless-Server (CSS)



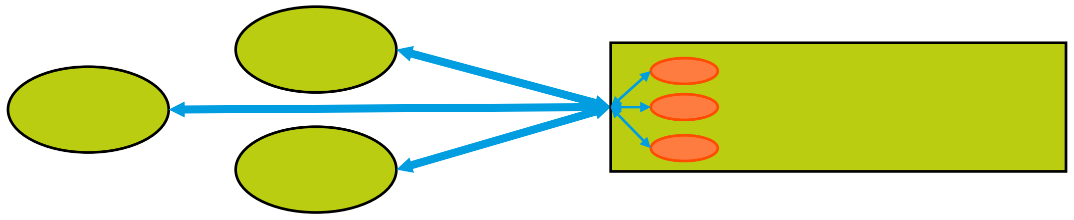
- Same as client/server
- Constraint: no session data on the server

Client-Cache-Stateless-Server (C$SS)



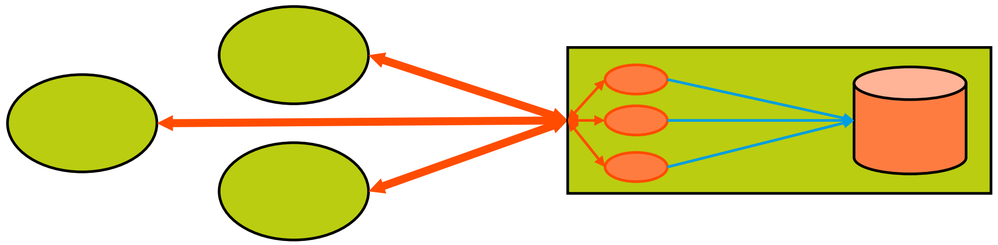
- Same as client-stateless-server
- Client can cache state data

Remote Session (RS)



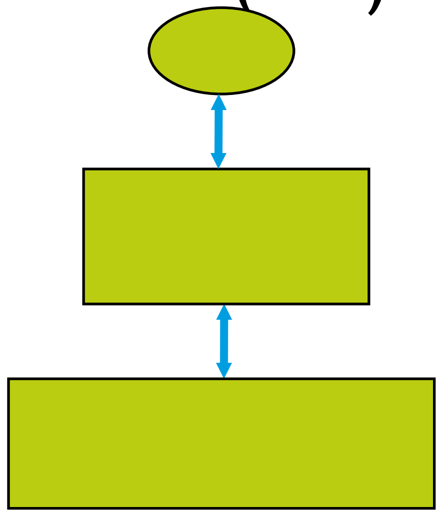
- each client initiates a session on server
- application state kept on server
- commands are used to exchange data or change session state
- flexible, interactive, easy to extend services
- scalability is a problem

Remote Data Access (RDA)



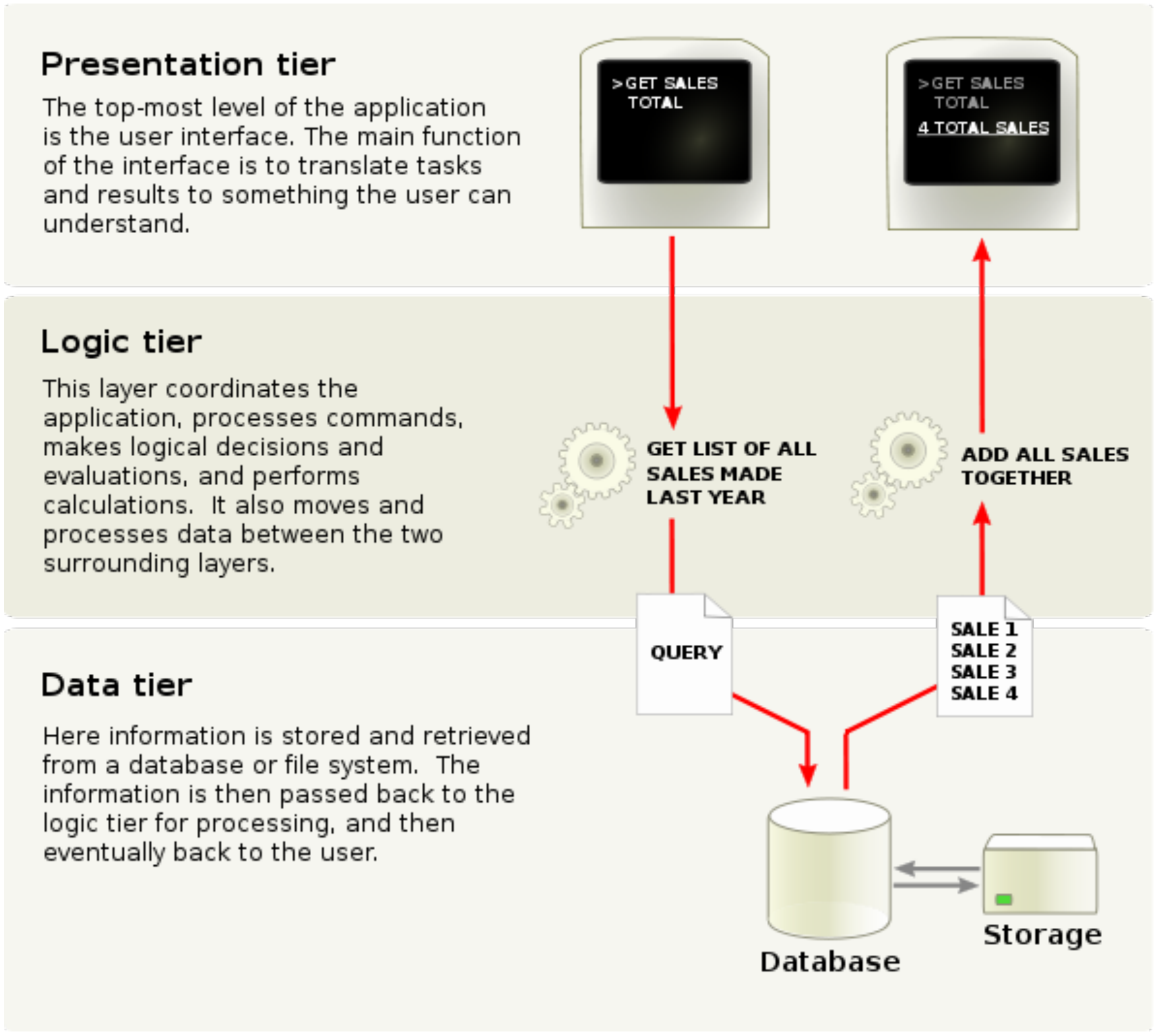
- Clients send database queries to remote server
- Server maintains per-client state for joins/trans.
- Client must know enough about data structure to build structure-dependent queries
- SQL commonly used to define query

Layered System (LS)



- each layer provides services to the layer above it and uses services of the layer below it
- Usage: layered-client-server
- 2-tiered, 3-tiered, or multi-tiered architectures

Layered System: 3-tier Architecture



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

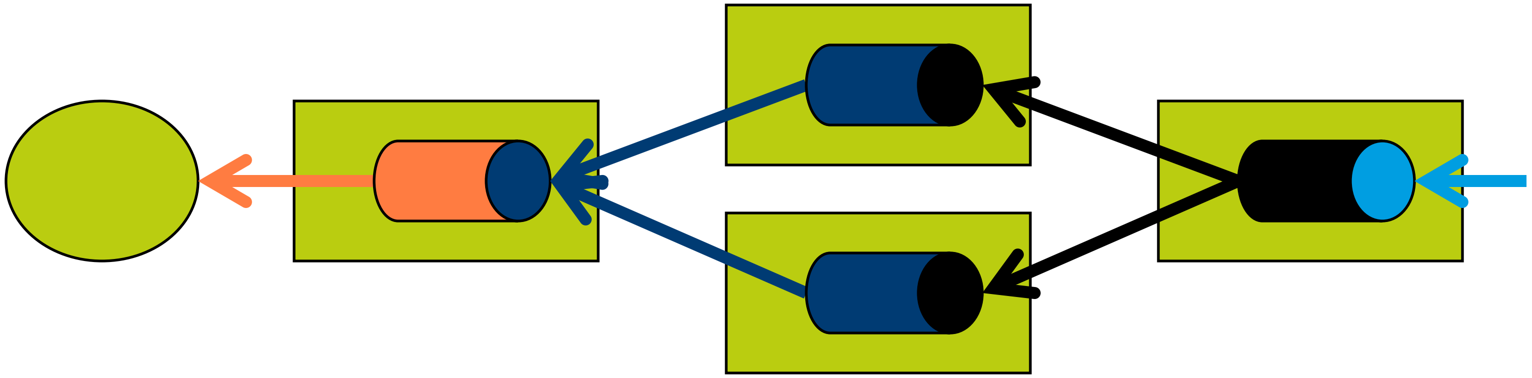
Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

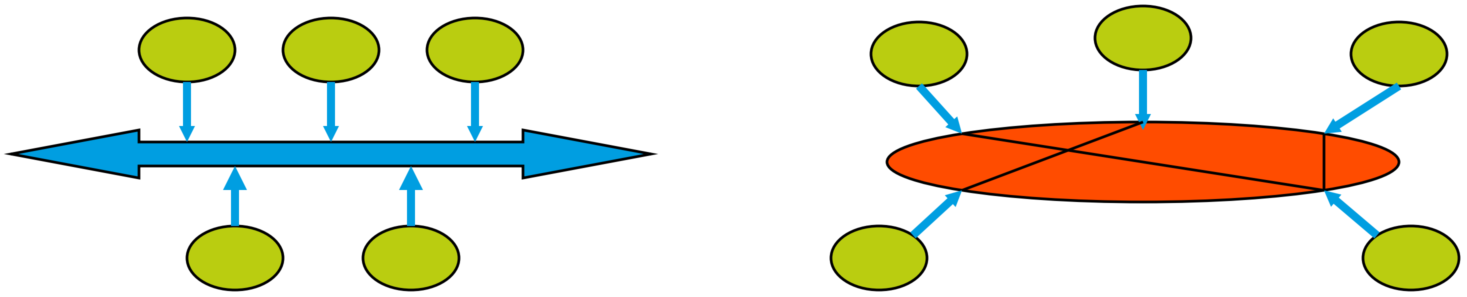
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

Pipe-and-Filter (PF)



- one-way data flow
- data stream is filtered through a sequence of components
- components do not need to know identity of peers

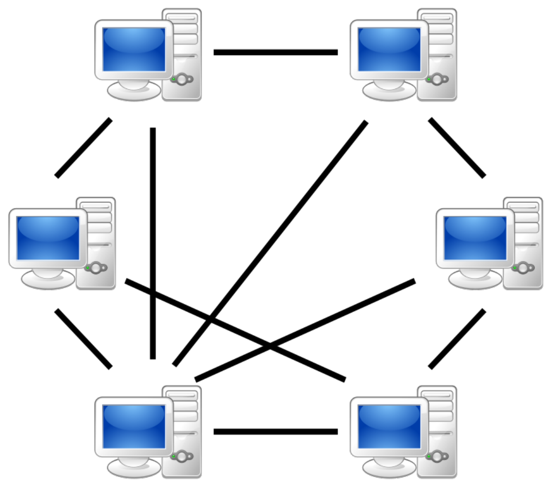
Event-based Integration



- components listen to a **message bus** or register with a broker
- components do not need to know identity of peers
- separation of concerns, independent evolution
- usually not designed for high-latency networks

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	-	±			++	+	+		++	++	+		
RR			++		+									+
$	RR		+	+	+	+								
CS					+	+	+							
LS			-		+		+				+		+	
LCS	CS+LS		-		++	+	++				+		+	
CSS	CS	-			++	+	+					+		+
C$SS	CSS+$	-	+	+	++	+	+					+		+
LC$SS	LCS+C$SS	-	±	+	+++	++	++				+	+	+	+
RS	CS			+	-	+	+					-		
RDA	CS			+	-	-						+		-
VM						±		+				-	+	
REV	CS+VM			+	-	±		+	+			-	+	-
COD	CS+VM		+	+	+	±		+		+		-		
LCODC$SS	LC$SS+COD	-	++	++	+4+	+±+	++	+		+	+	±	+	+
MA	REV+COD		+	++		±		++	+	+		-	+	
EBI				+	- -	±	+	+		+	+	-		-
C2	EBI+LCS		-	+		+	++	+		+	++	±	+	±
DO	CS+CS	-		+			+	+		+	+	-		-
BDO	DO+LCS	-	-				++	+		+	++	-	+	

Peer-to-Peer Architectures (P2P)



- Distributed participants:
  - make a portion of their resources (CPU, disk, network bandwidth) directly available to other participants
  - no need for central coordination instances (such as servers or stable hosts)
  - Peers are both suppliers and consumers of resources
- Examples: Napster, Skype, and BitTorrent

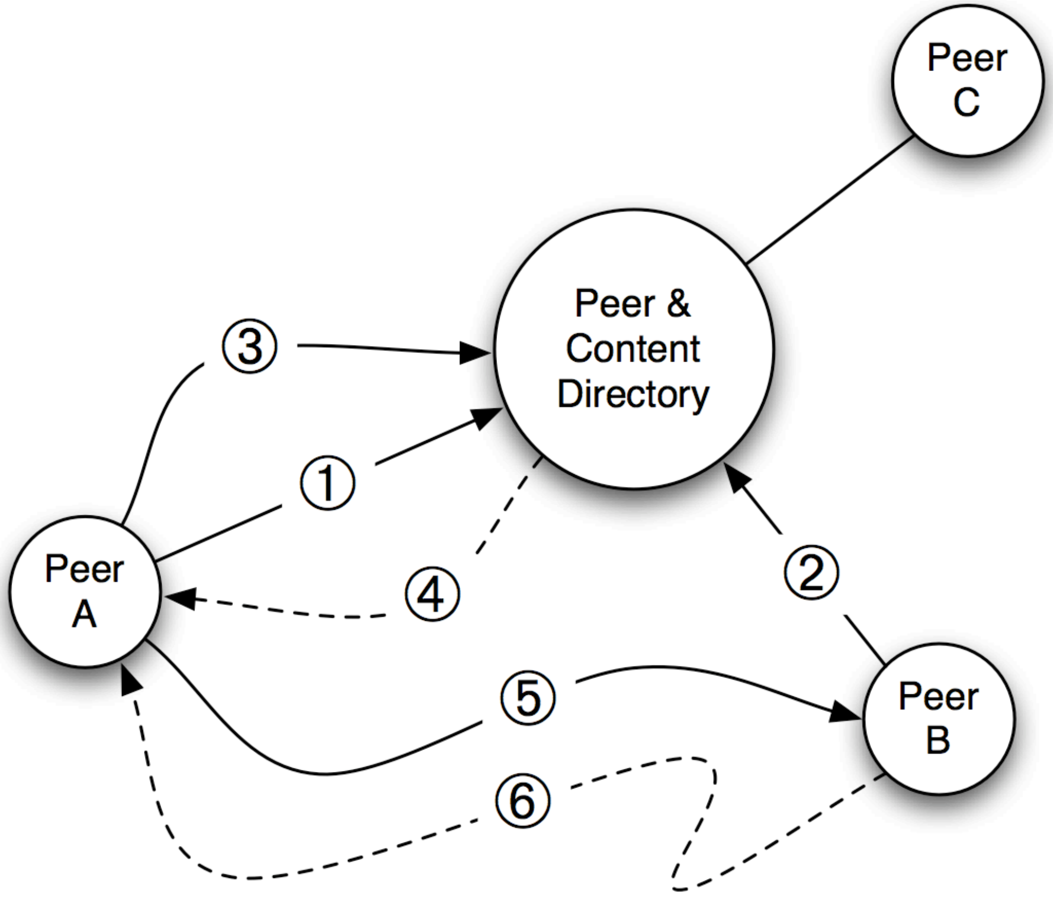
Peer-to-Peer Style

- Peers: independent components, having their own state and control thread.
- Connectors: Network protocols, often custom.
- Data Elements: Network messages
- Topology: Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Highly robust in the face of failure of any given node. Scalable in terms of access to resources and computing power.

Napster (Hybrid: CS and P2P)

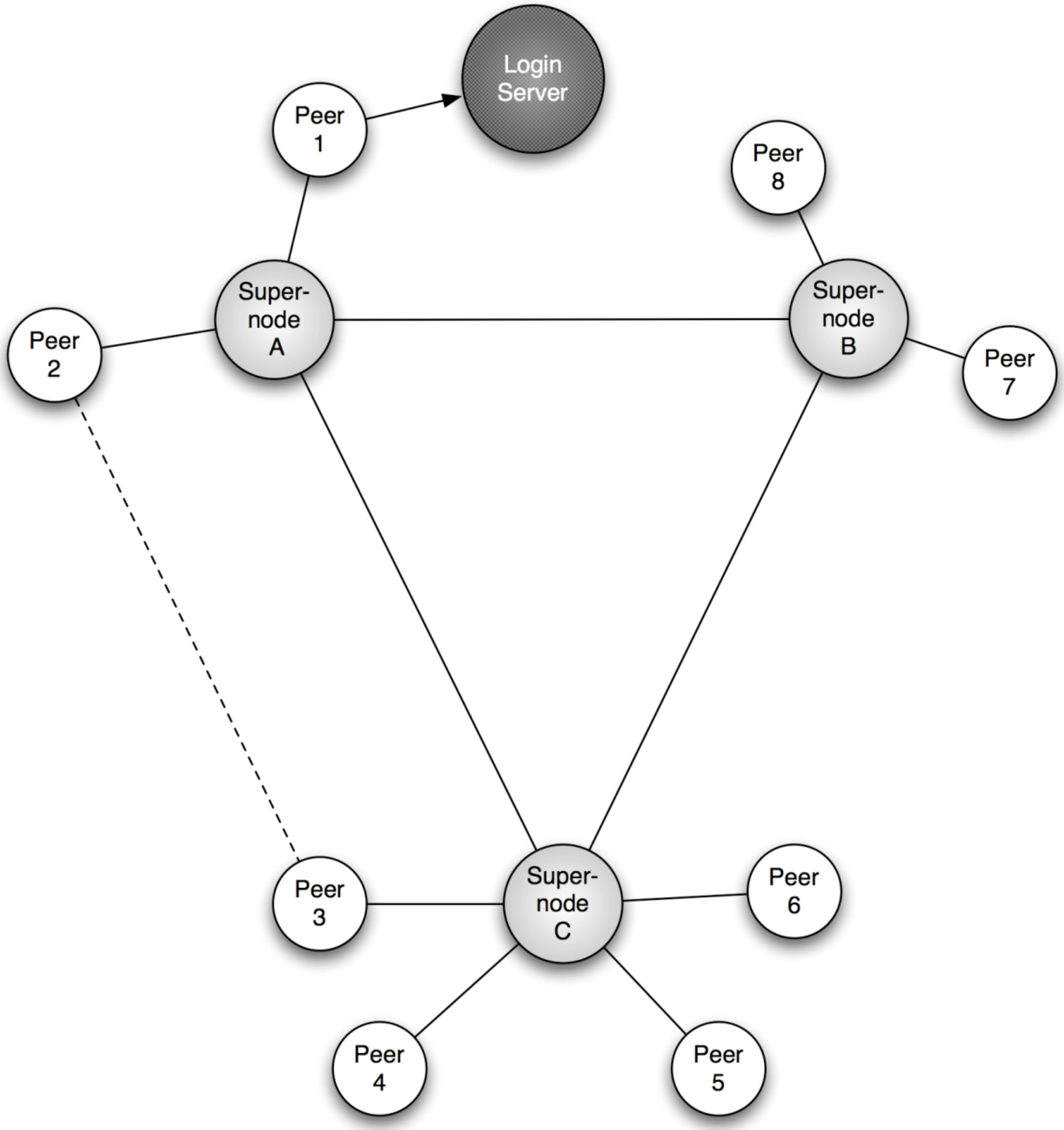
Limitations?

1. A highly desired song: server becomes overloaded
2. Server goes down: no communication between peers.

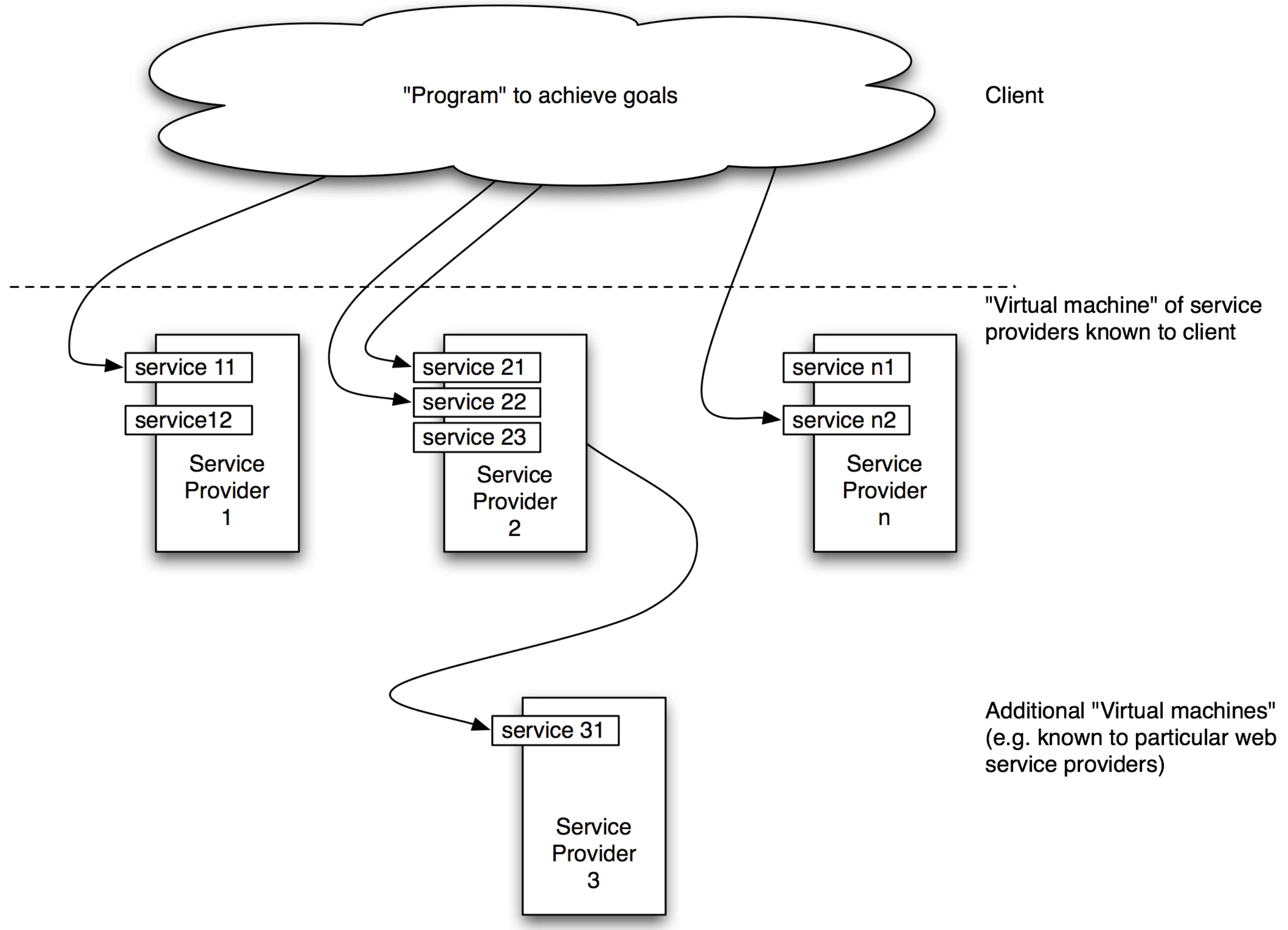


Skype

- Any peer can become a super node by Skype
- Does not suffer from scalability or discovery problems (Napster)
- Protocol is closed, proprietary -> privacy
- Down side: high bills for super nodes

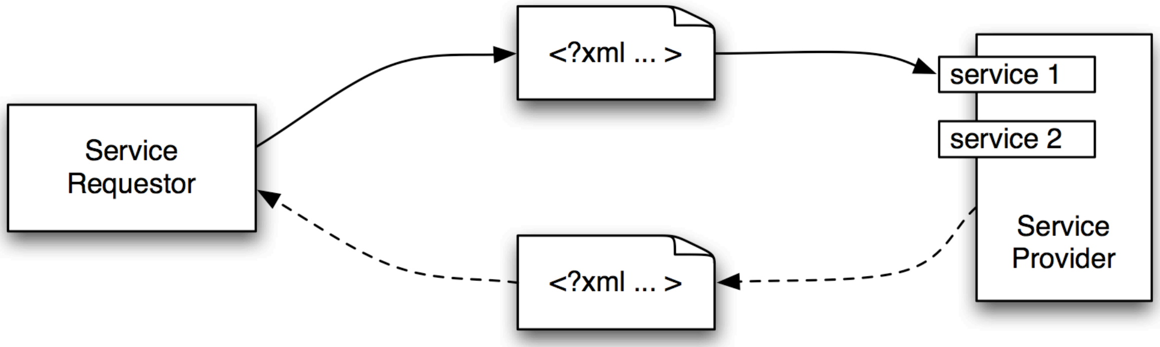


Service Oriented Architecture (SOA)


"Program" to achieve goals
Client
"Virtual machine" of service providers known to client
service 11
service12
Service Provider 1
service 21
service 22
service 23
Service Provider 2
service n1
service n2
Service Provider n
service 31
Service Provider 3
Additional "Virtual machines" (e.g. known to particular web service providers)

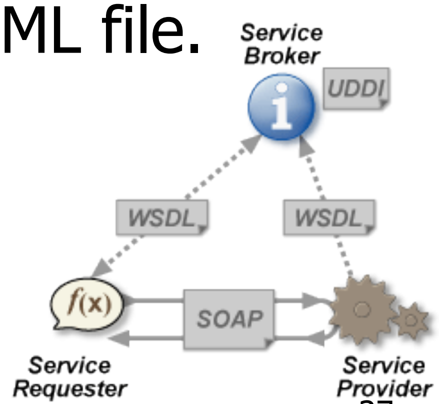
loosely-integrated suite of *services* that can be used within multiple business domains.

Web Services



- Services described in WSDL
    - Available operations on typed data to/from service
    - Serves as the interface (contract) between requester and provider

- SOAP: used for structuring the exchanged XML file.



In-class exercise: Architecture of the Web

- Draw the architecture of the **Web** using two or more architectural styles.

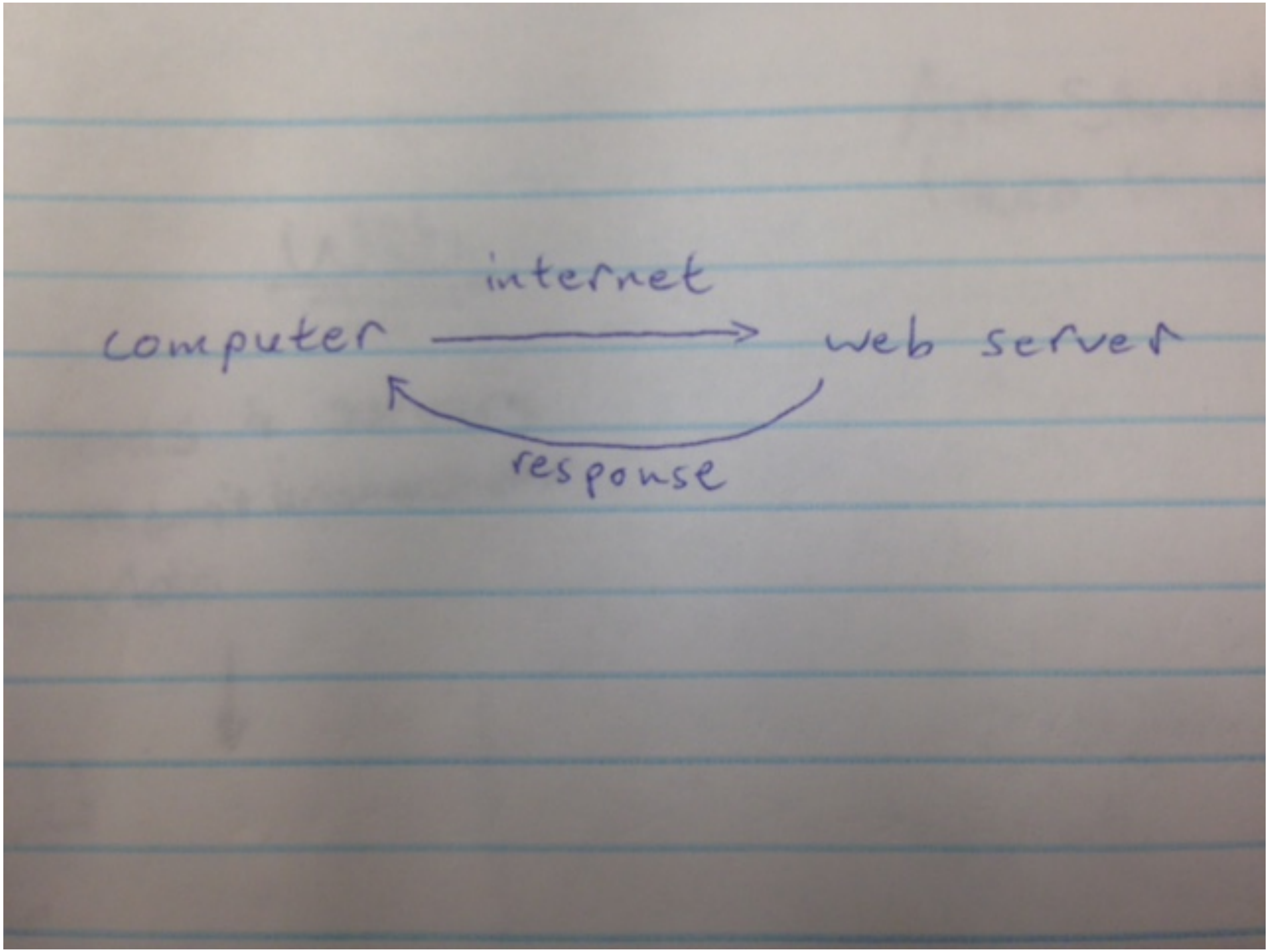
- Which elements?
- Which constraints?
- Which properties?

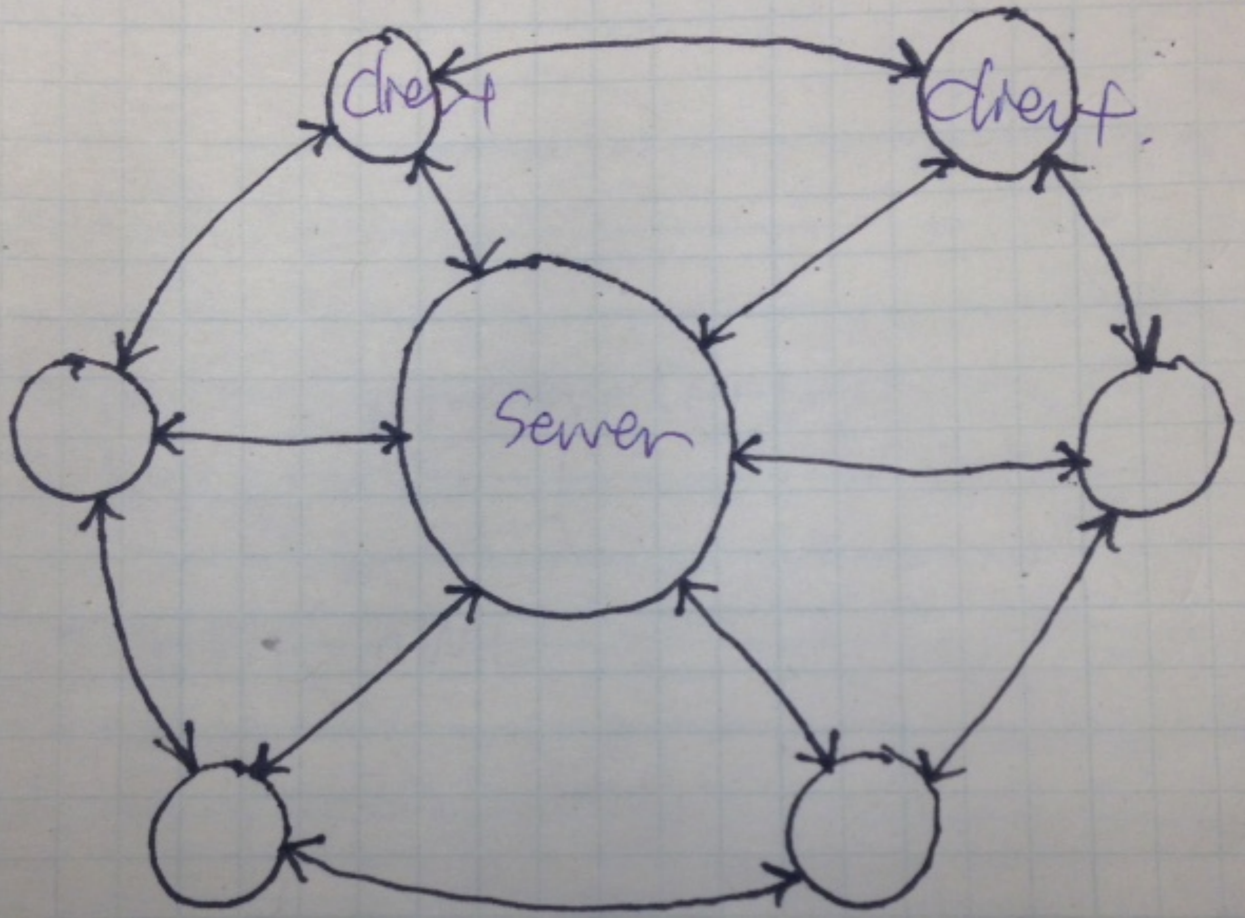
Facebook helps you connect and share with the people in your life.

It's free and anyone can join

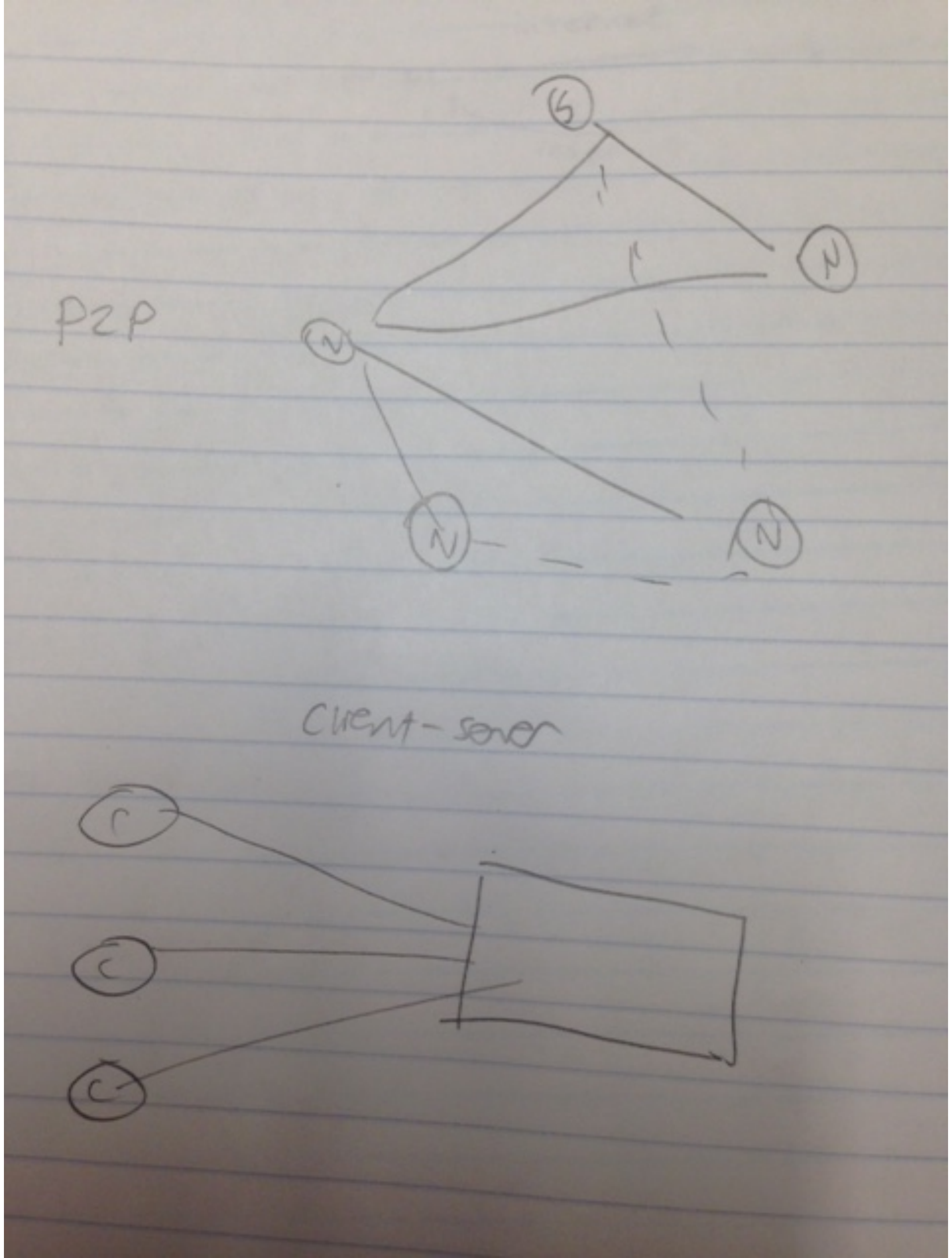
By clicking Sign Up, you are indicating that
to the Terms of Use and Privacy Policy.

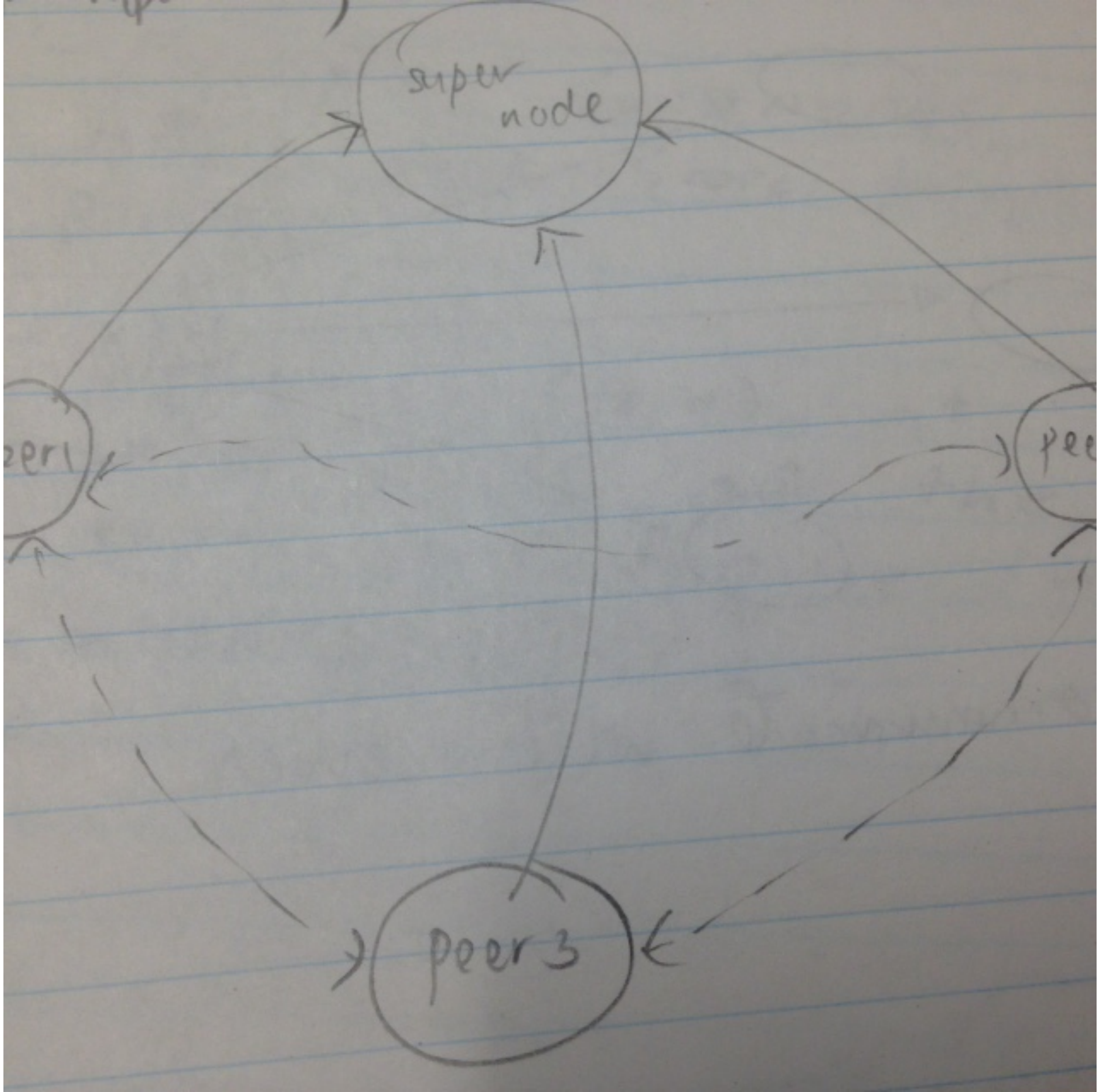
SOME EXAMPLES (PRODUCED BY YOU)

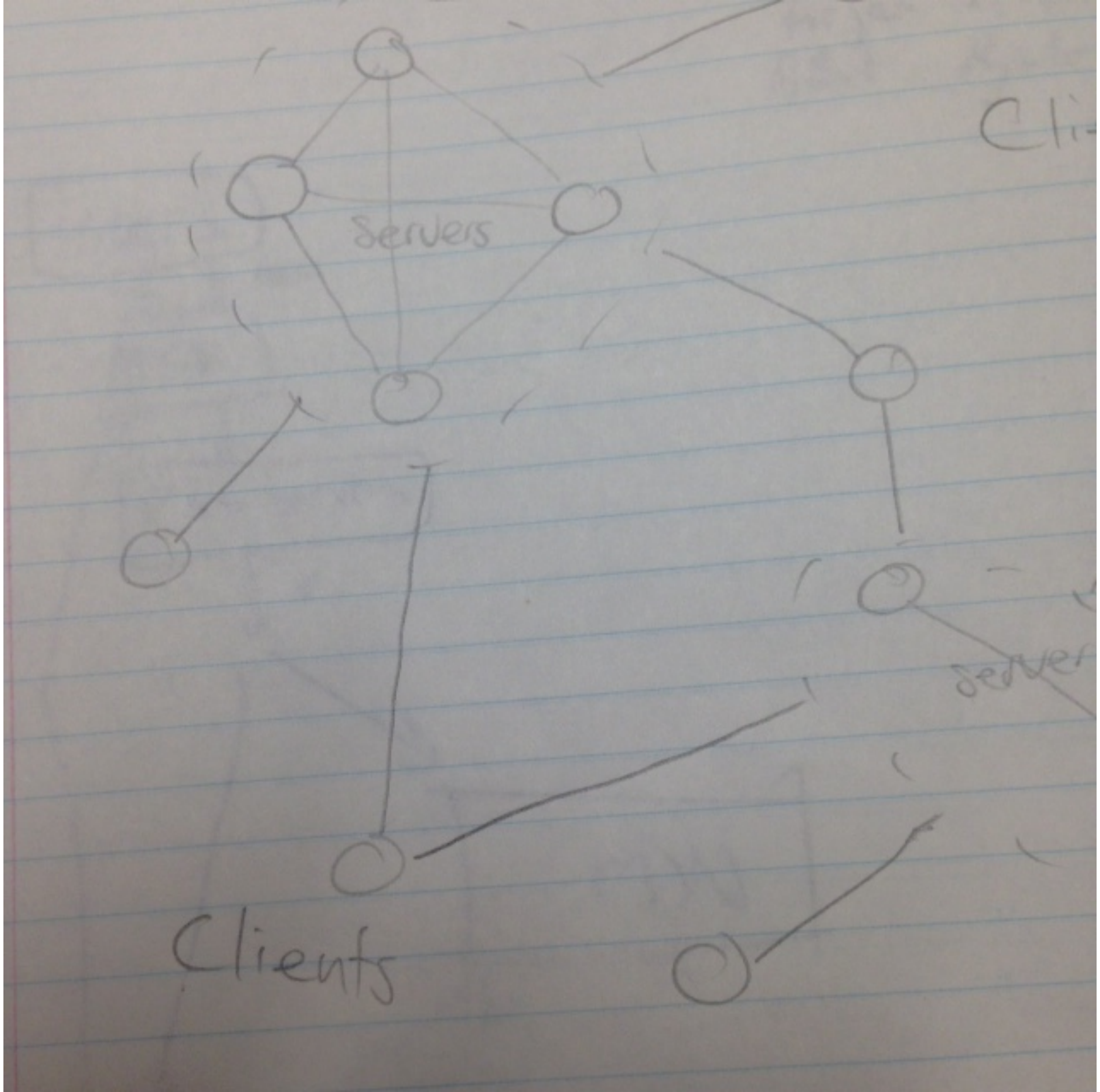

internet
computer
web server
response



Client Server && Peer to Pe

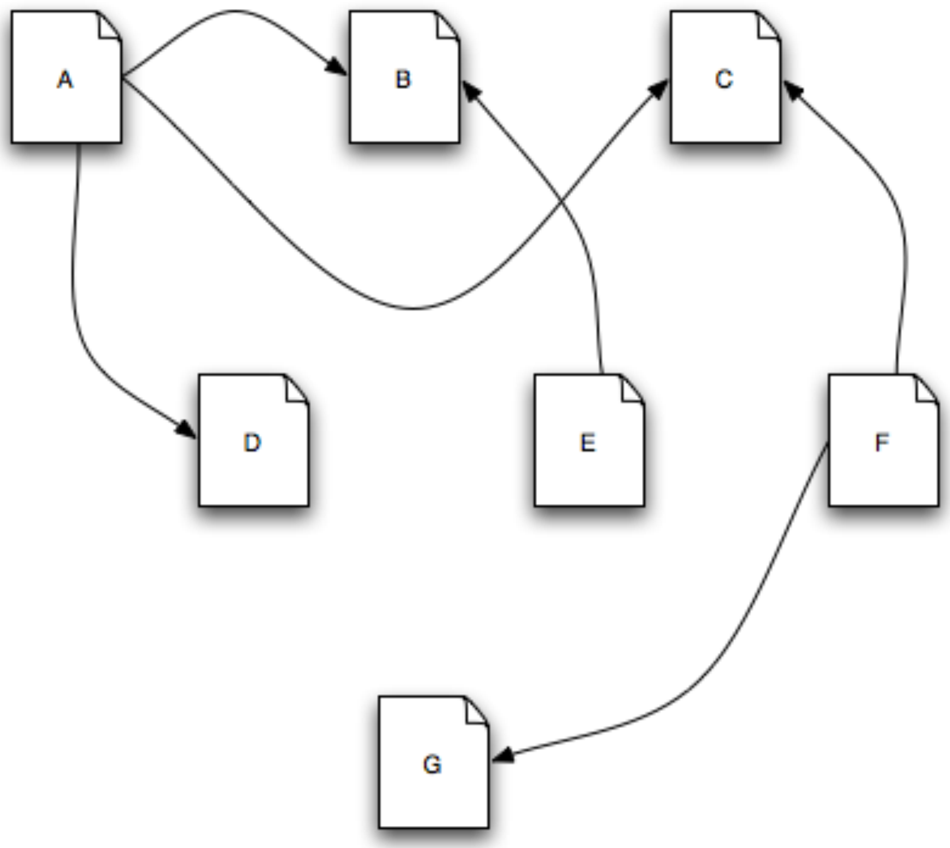

P2P
S
N
N
N
N
Client-server
C
C
C


super node
peer1
peer3


Servers
Cli
Server
Clients

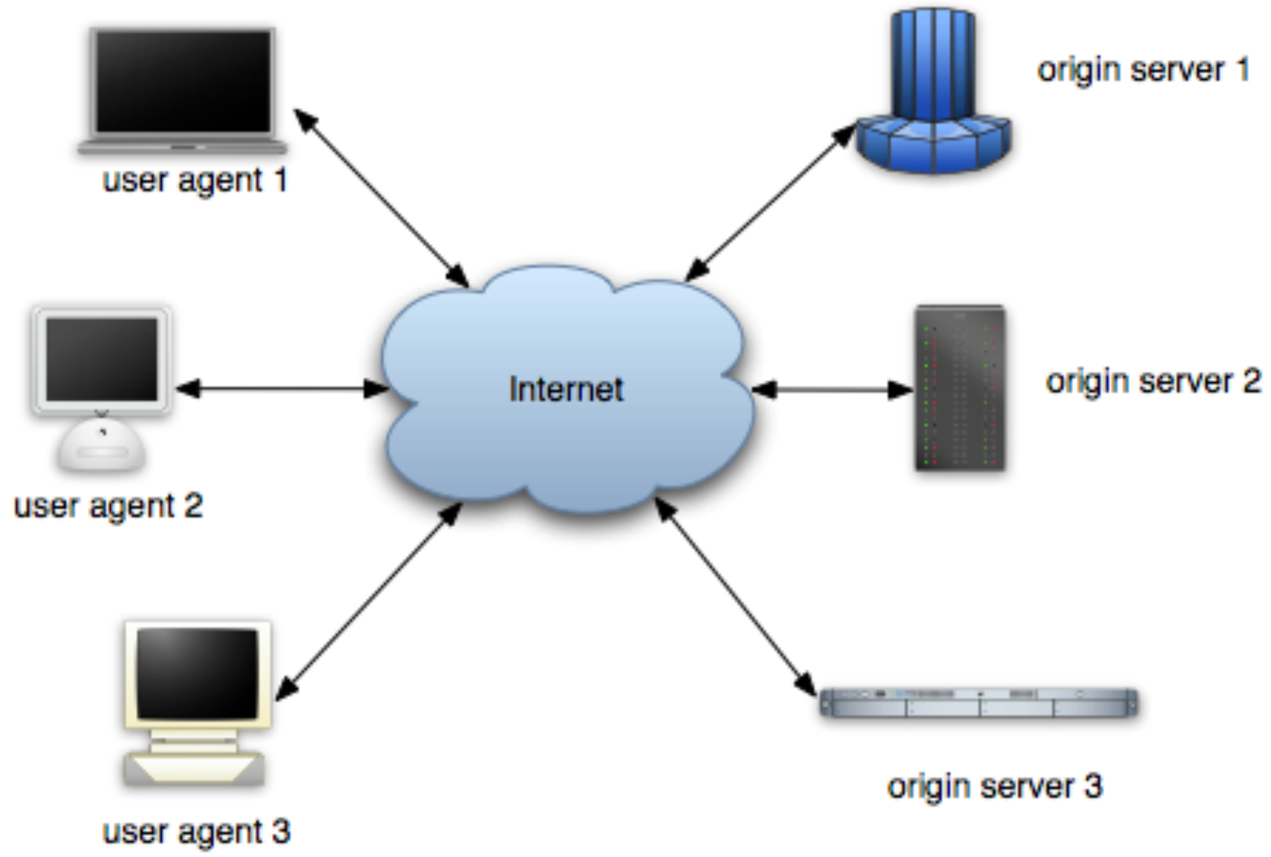
Architecture in Action: WWW

Is this it?



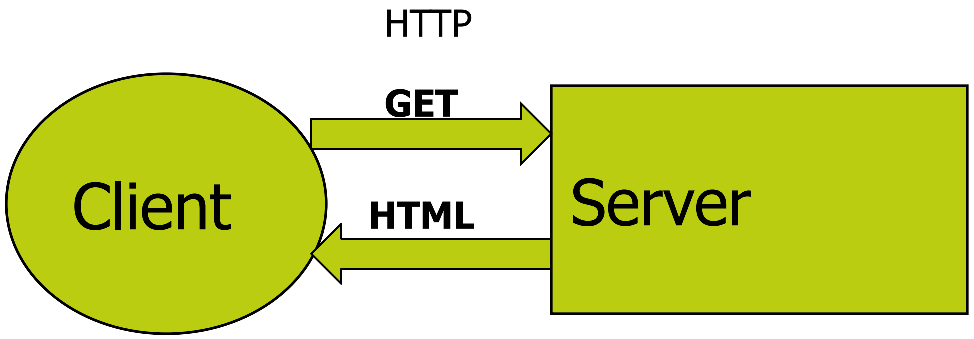
Architecture in Action: WWW

and this?



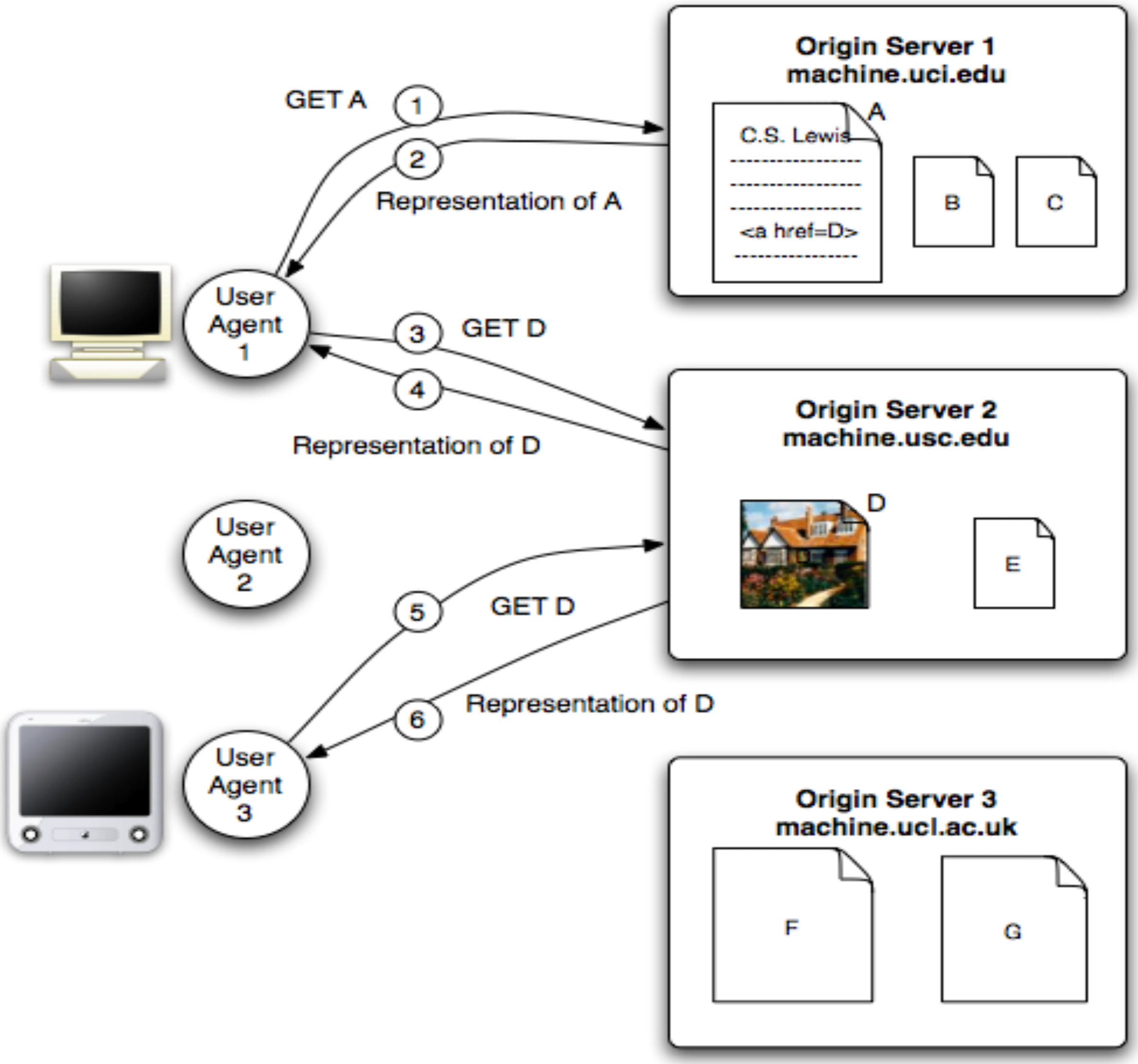
Architecture in Action: WWW

and this?

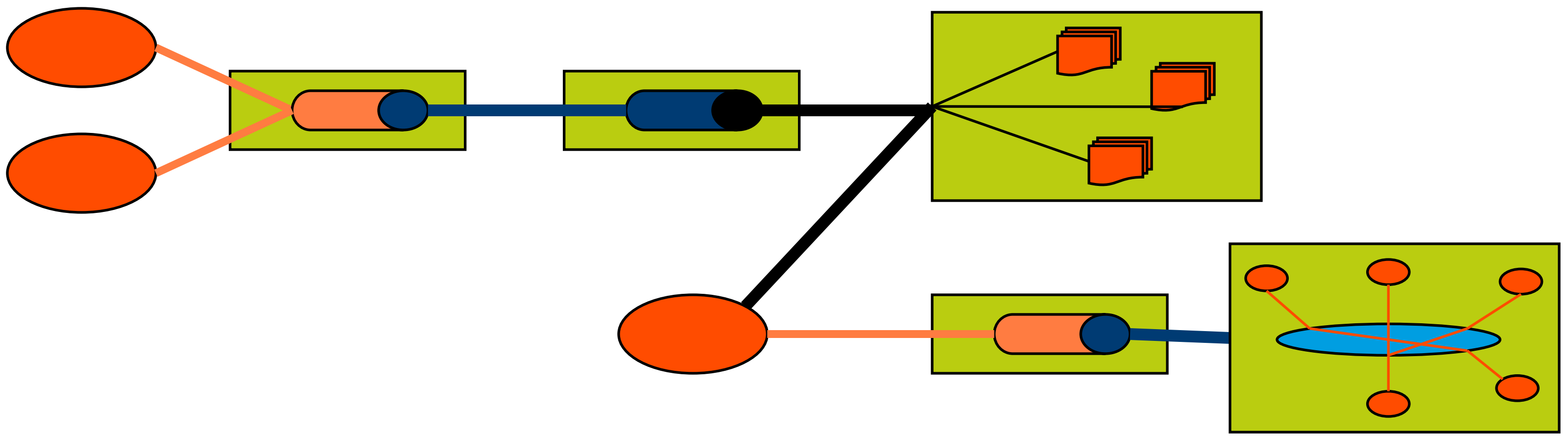


Architecture in Action: WWW

How
about this?



REST: Web Architectural Style



- optimized for transfer of typed data streams

REST: Web Architectural Style

- REpresentational State Transfer (Roy Fielding in 2000)
  - A model of how the Web should work
- The Web is a *distributed hypermedia application*
- Architecture of the Web is totally separate from the code
- Derived by combining various architectural styles

Web Architectural Properties

architectural constraints + web elements
= desired web properties

- Simplicity
- Scalability
- Modifiability
- Performance

- Visibility (monitoring)
- Portability
- Reliability

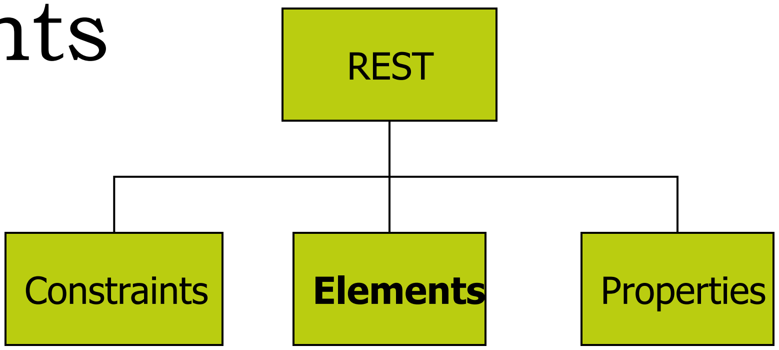
REST is not a set of laws or even a specification. It's a style.

REST Notions

REST revolves around five fundamental notions:

- Resources
  - Names of all ECE students
- Representation of a resource
  - HTML, XML, JSON
- Communication to obtain/modify representations
  - HTTP: Get, Post, Delete, Put (think CRUD)
- Web "page" as an instance of application state
- Engines to move from one state to the next
  - Browser, Spider

REST — Data Elements



- Resource
  - Key information abstraction
- Resource ID
  - URI
- Representation
  - HTML document, JPEG image
- Representation metadata
  - media type, last-modified time

REST: Resources

- The key abstraction of information is a resource, named by an identifier such as URL (constraint)
  - http://example.com/products/books
- A resource can be anything that has identity
  - a document or image
  - a service, e.g., “today’s weather in Seattle”
  - a collection of other resources
- Resources have state that may change over time
- Resources expose a uniform interface (constraint)
  - System architecture simplified, visibility improved. Encourages independent evolvability of implementations.

REST: Representations of a Resource

- The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes.
- On request, a resource may transfer a representation of its state to a client
  - Necessitates a client-server architecture (constraint)
- A client may manipulate resources through representations: POST, DELETE, PUT (constraint)

REST: Representations of a Resource

- Hypermedia-aware media types
  - HTML, XML, etc.
- Representations should be cachable (constraint)
- Representations returned from the server should link to additional application state.
  - Hypermedia as the engine of application state (constraint)

REST is Stateless

- Stateless interactions (constraint)
  - Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server
- Uniform interface + Stateless + Self-descriptive = Cacheable (constraint)
- Cacheable necessitates a layered-system (constraint)

REST Constraints and Properties

Constraint	**Property**
Client-Server	Separation of concerns: simplicity, scalability, evolvability
Stateless	Scalability, reliability, visibility
Cacheable	Scalability, performance
Uniform Interface	Simplicity, evolvability, visibility
Layered System	Scalability, security, legacy integration
Identification of Resources	Simplicity, scalability. Required by uniform interface
Manipulation via representations	Evolvability. Required by uniform interface
Self-descriptive messages	Required by stateless, cachebale, uniform interface, and layered system
Hypermedia as the engine of application state	Scalability, reliability, evolvability, performance. Required by uniform interface
Code on demand (optional)	Evolvability

Derivation of REST

Key choices in this derivation include:

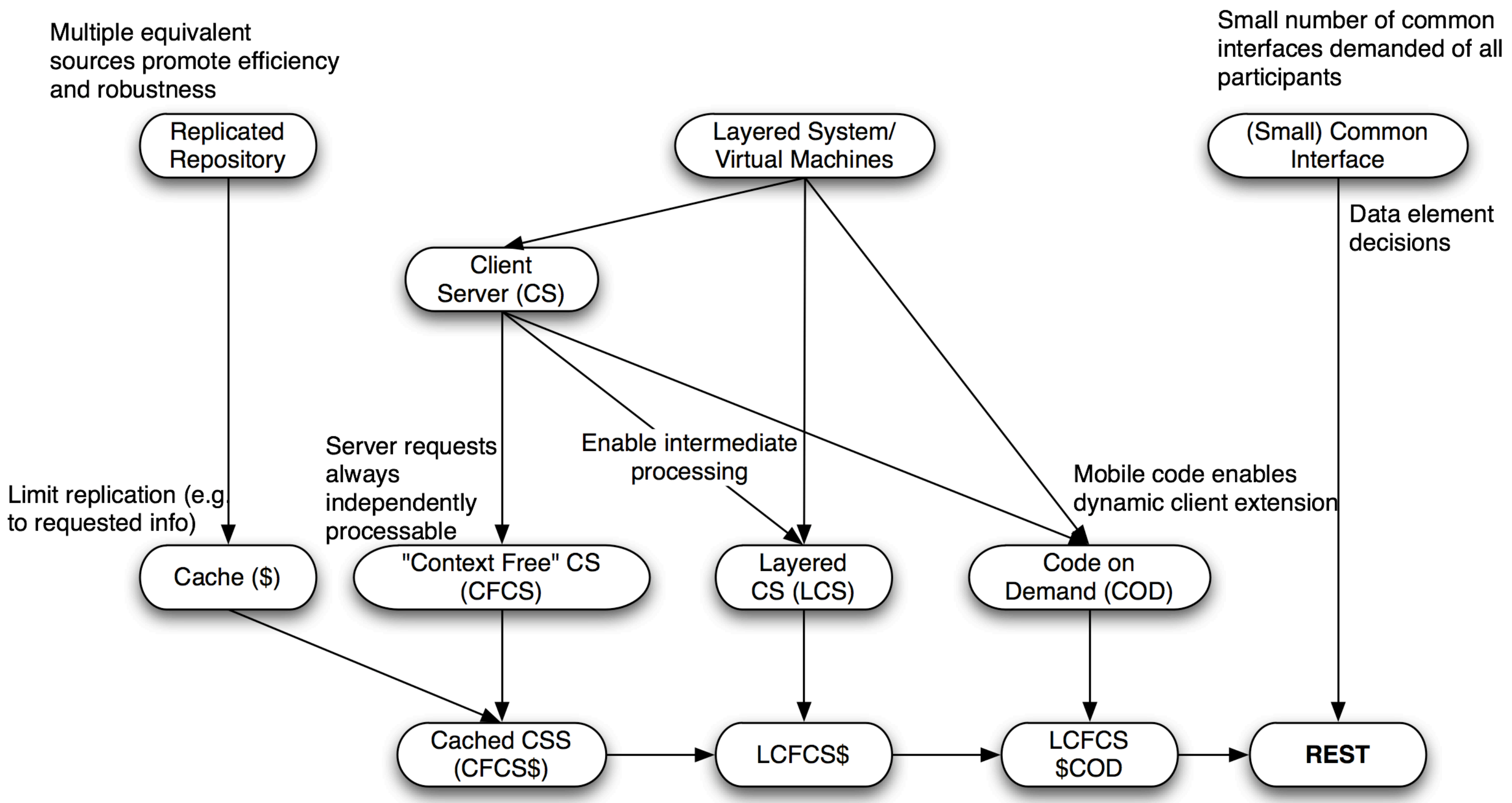
- Layered Separation: to increase efficiencies, enable independent evolution of elements of the system, and provide robustness;

- Common interface: universally understood operations with extensibility.

- Stateless: helps scalability.

The derivation is driven by the application (!)

Derivation of REST (cont’d)


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

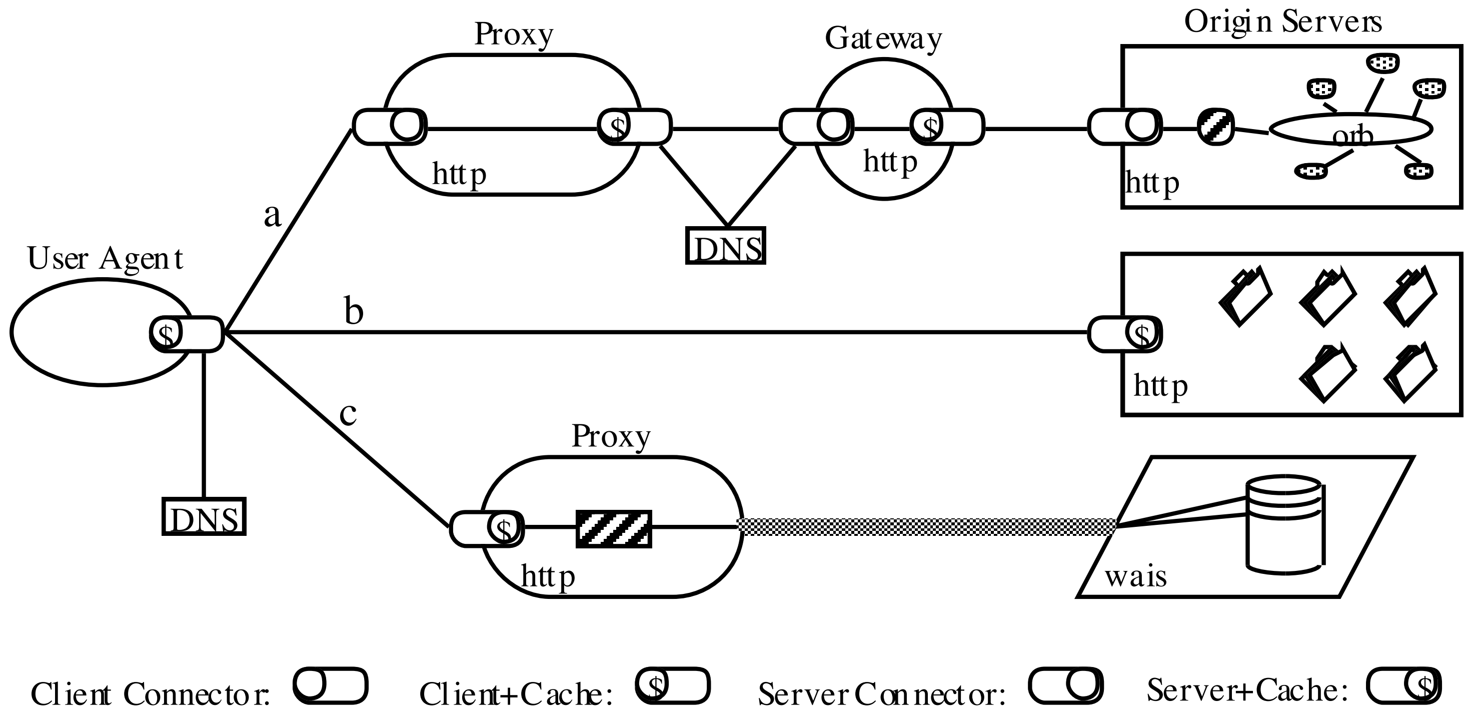
REST — Connectors

• Client	libwww
• Server	Apache API
• cache	browser cache
• resolver	bind (DNS lookup library)
• Tunnel	SSL after HTTP CONNECT

REST — Components

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

A REST-based architecture

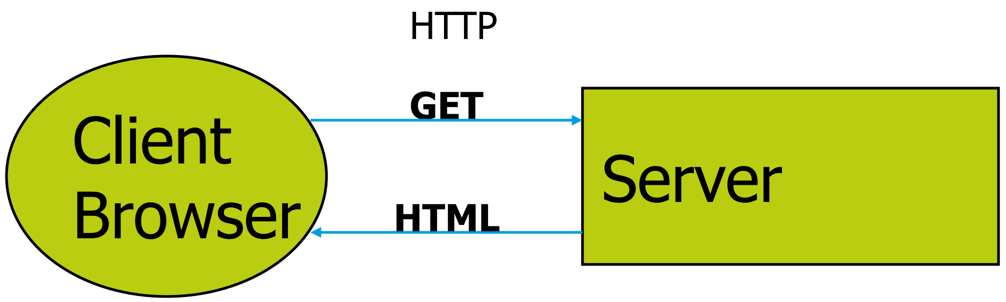


REST: Final Thoughts

- **RE**presentational **S**tate **T**ransfer
- Constrained Client/Server with focus on data elements
- Style of "classical" web architecture: page sequence (multi-page)
- Modern Web diverges from style
  - e.g., Ajax, mashups
  - Comet (push)

The Modern Web Architecture

Classical Web Applications



Every state change implies:

- Client sends request to Server
- Server builds entire page
- Server sends new page to Client
- Client refreshes the entire page

Based on: **Multi Page Interface Model**

Multi Page Interface Shortcomings

MPI’s Architectural Shortcomings:

- Low lever of user interactivity
- Redundant data transfer
- High user-perceived latency
- Client is passive

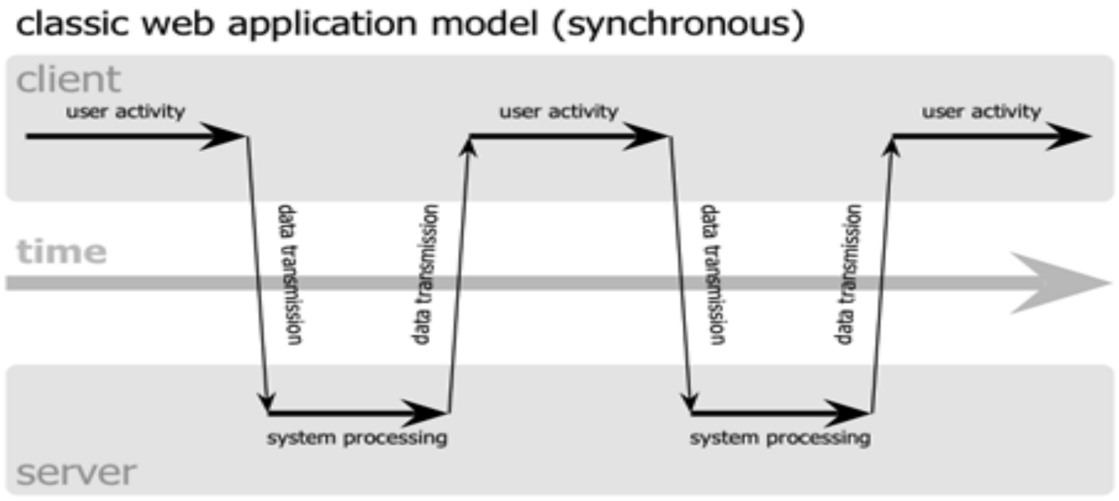
Web 2.0

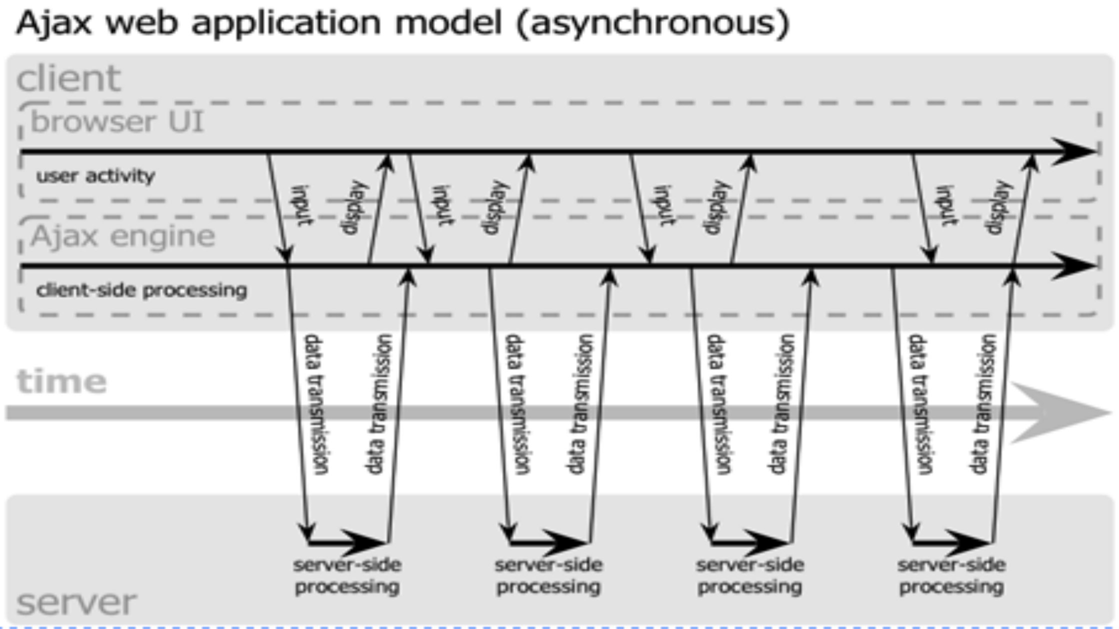
Rich user interaction on the Web

- Google Maps / Gmail, Docs, Yahoo! Mail

Ajax: “Asynchronous JavaScript And XML”

- Asynchronous server communication using XMLHttpRequest
- Web Standards based
  - HTML, CSS, XML
  - (excludes Flash, Applets, …)
- Dynamic display using Document Object Model (DOM)
- JavaScript binds everything together


classic web application model (synchronous)
client
user activity
user activity
user activity
data transmission
data transmission
data transmission
data transmission
time
system processing
system processing
server


Ajax web application model (asynchronous)
client
browser UI
user activity
input
display
Ajax engine
client-side processing
data transmission
data transmission
time
server-side processing
server

Software Architecture Domain (overview)

- Software Architecture
  - Network-based Architecture
    - Client/Server Architecture
      - Web-based Architecture
        - **Ajax-based Architecture**

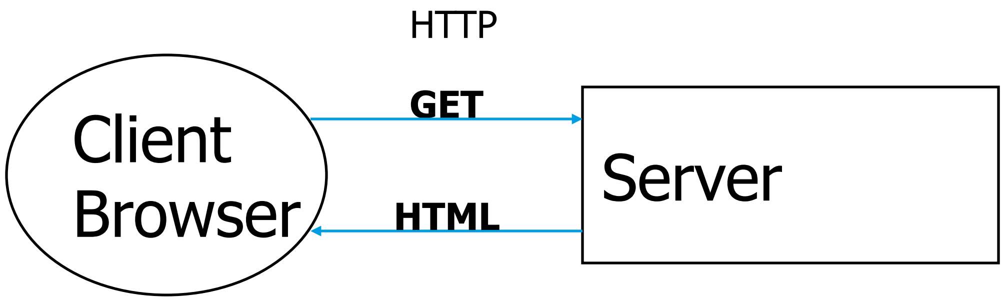
Modern Web Architecture

- What would it look like?

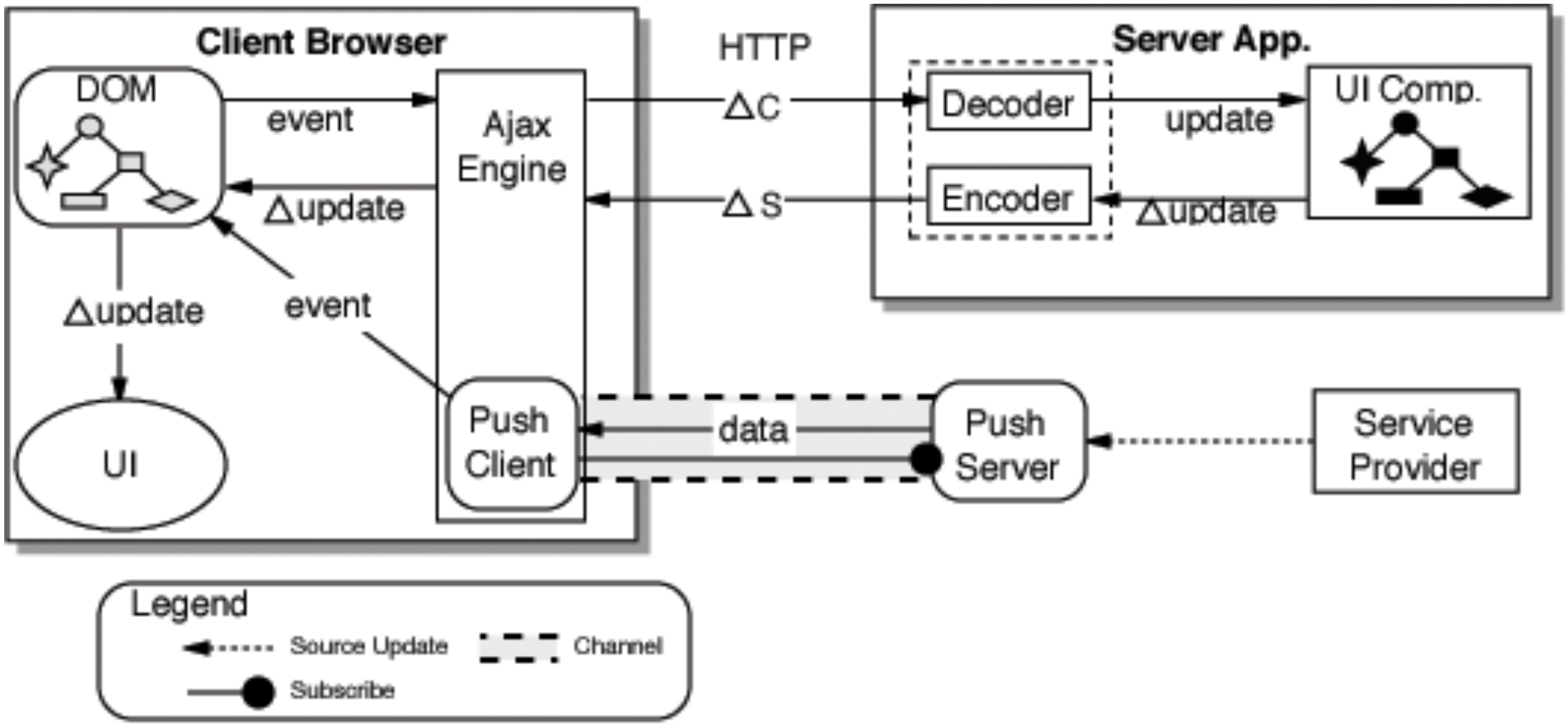
REST versus Ajax

Suited for large-grain hypermedia data transfers	Requires small data interactions
Resource-based	UI component-based
Hyper-linked	Action- Event-based
Synchronous request-response	Asynchronous interaction
Stateless	Stateful
Cache-based	Real-time data retrieval

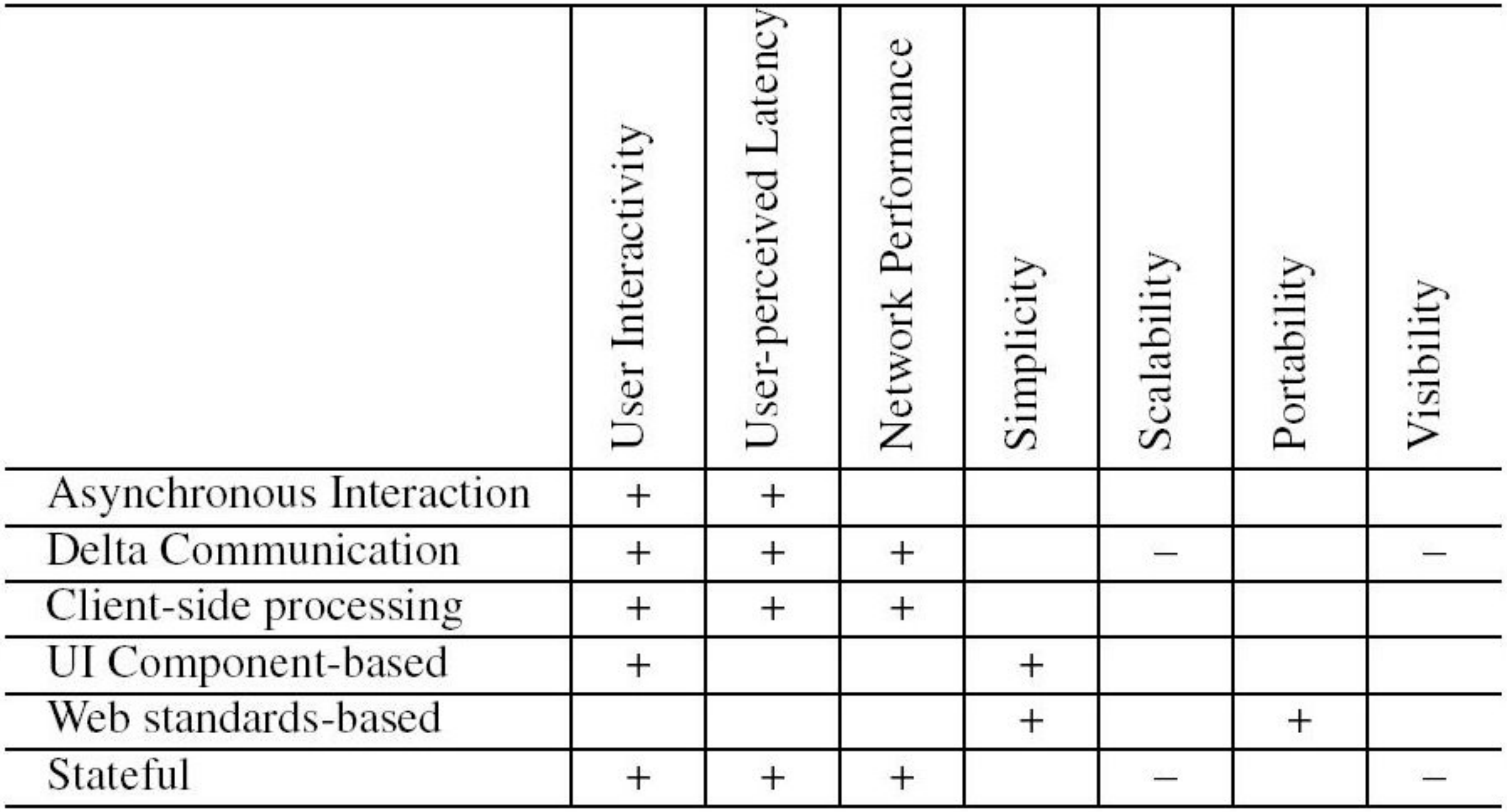
Classical Architecture



Ajax-based Architecture



Constraints and Induced Properties



	User Interactivity	User-perceived Latency	Network Performance	Simplicity	Scalability	Portability	Visibility
Asynchronous Interaction	+	+					
Delta Communication	+	+	+		–		–
Client-side processing	+	+	+				
UI Component-based	+			+			
Web standards-based				+		+	
Stateful	+	+	+		–		–

Takeaways

- A great architecture is the ticket to success
- A great architecture reflects deep understanding of the problem domain
- A great architecture probably combines aspects of several simpler architectures
- Develop a new architectural style with great care and caution. Most likely you don’t need a new style.