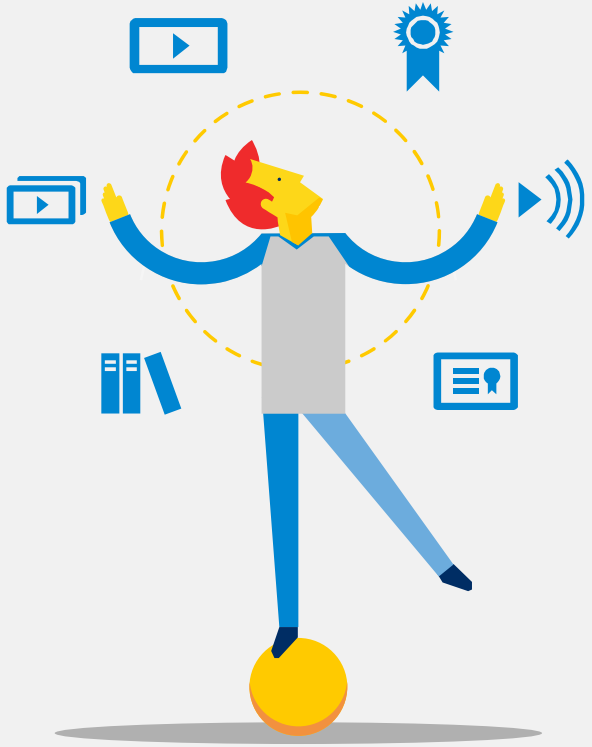
Microservices and Messaging

Clemens Vasters
Architect, Azure Messaging
@clemensv

Microsoft



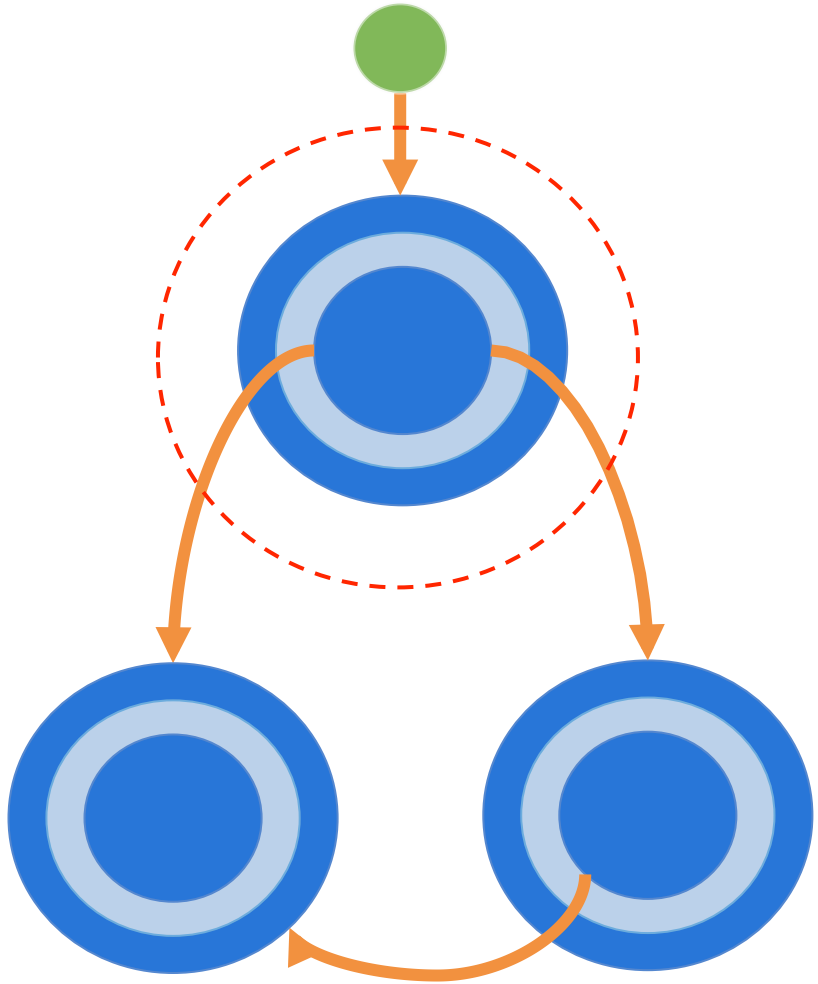
@clemensv

Clemens Vasters

- I'm an Architect working on Microsoft's Azure Compute and Messaging team, working on Service Bus, and Event Hubs for most of the last 10 years. I've also had a hand in creating services like Azure Notification Hubs, and the Azure IoT Hub, and several aspects of the foundational "behind the scenes" pieces of Azure. I am engaged in Microsoft's messaging standardization efforts for AMQP (and MQTT) and in industrial communication standards like OPC UA, where I also sit on the technical advisory council.
- Before joining Microsoft in 2006, I co-owned a developer education and architectural consulting firm focusing on Service Oriented Architecture and Distributed Systems.

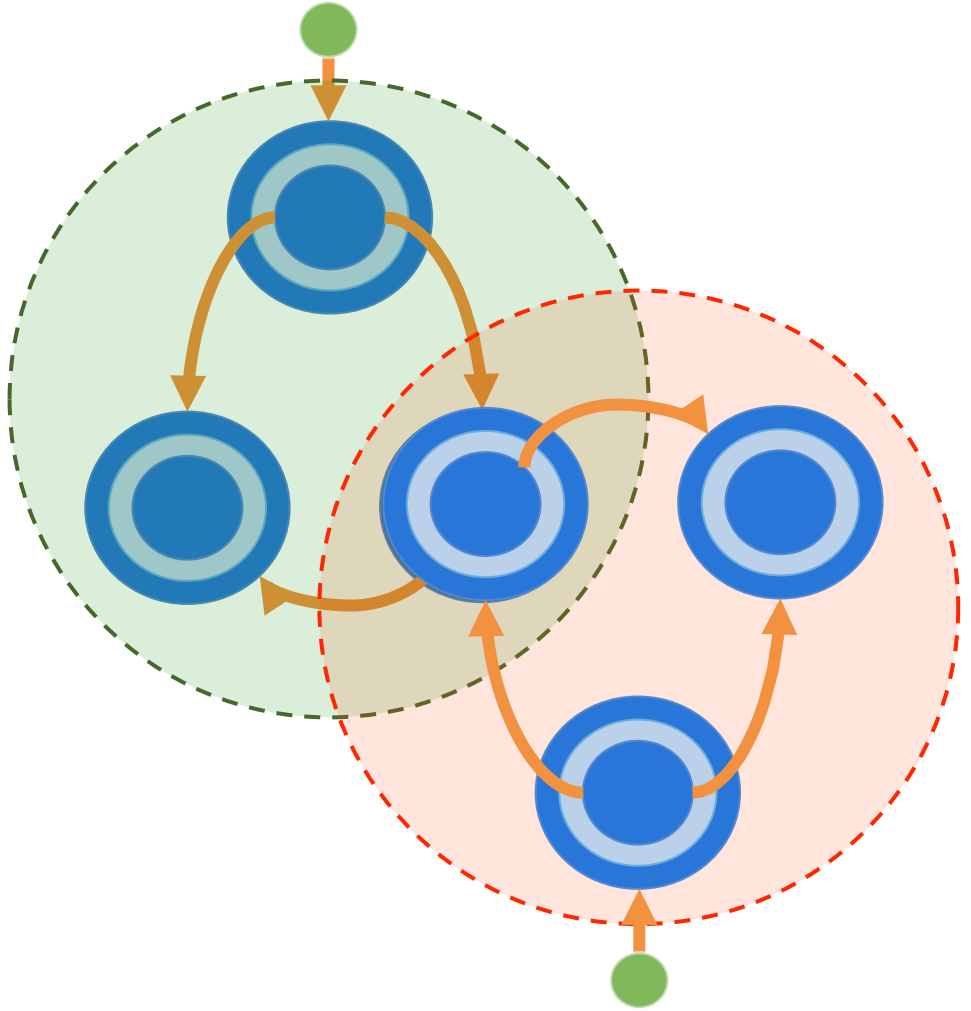
A “Service” is software that …

- … owned, built, and run by an organization
- … is responsible for holding, processing, and/or distributing particular kinds of information within the scope of a system
- … can be built, deployed, and run independently, meeting defined operational objectives
- … communicates with consumers and other services, presenting information using conventions and/or contract assurances
- … protects itself against unwanted access, and its information against loss
- … handles failure conditions such that failures cannot lead to information corruption



System

- A system is a federation of services and systems, aiming to provide a composite solution for a well-defined scope.
- The solution scope may be motivated by business, technology, policy, law, culture, or other criteria
- A system may appear and act as a service towards other parties.
- Systems may share services
- Consumers may interact with multiple systems



“Service” does not imply…

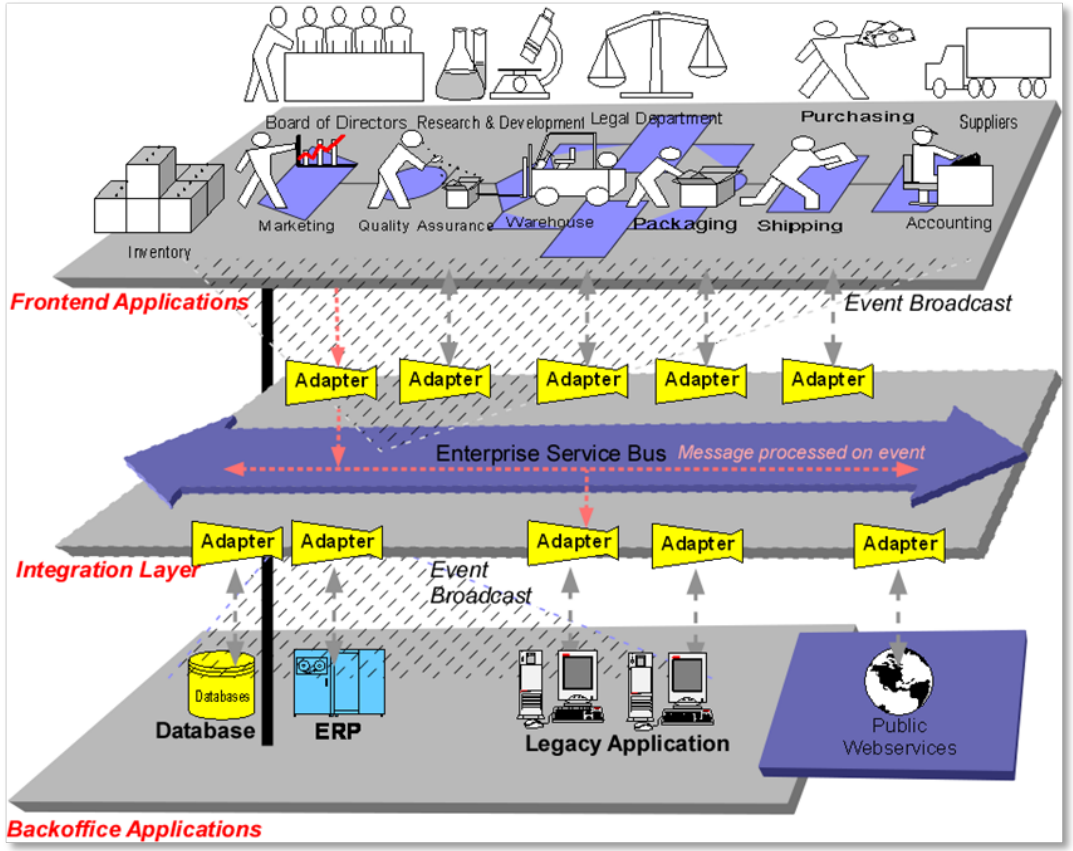
- … “Cloud”
- … “Server”
- … “ESB”
- … “API”
- … XML
- … JSON
- … REST
- … HTTP
- … SOAP
- … WSDL
- … Swagger
- … Docker
- … Mesos
- … Svc Fabric
- … Zookeeper
- … Kubernetes
- … SQL
- … NoSQL
- … MQTT
- … AMQP
- … Scale
- … Reliability
- … “Stateless”
- … “Stateful”
- … OAuth2
- … OpenID
- … X509
- … Java
- … Node
- … C#
- … OOP
- … DDD
- etc. pp.

Principles

of Service-Based Architecture

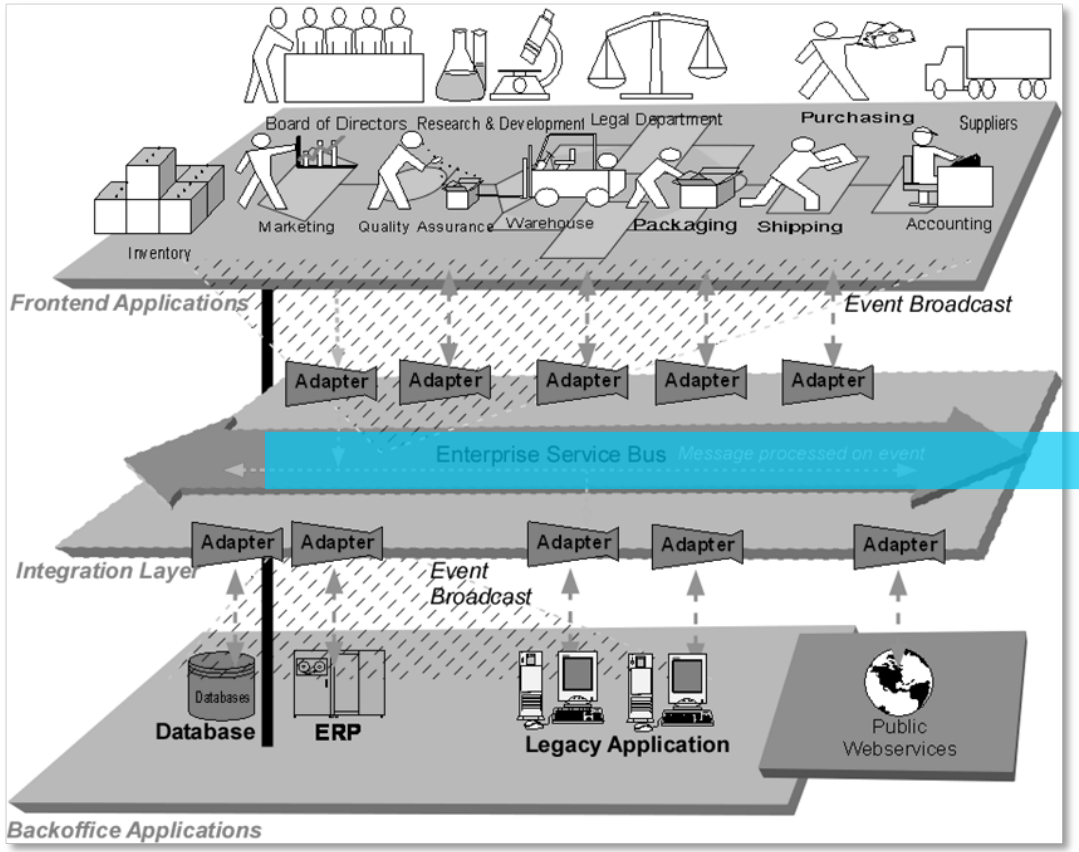
are independent of implementation choices.

About that “API Gateway” (nee ESB)



http://en.wikipedia.org/wiki/File:ESB.png

The Bus that's a Hub



The reality of the "Bus" is a Hub
n-way cluster with centralized repository

Challenges: Scale-Out, Connectivity, HA/Geo/ DR, Repository Complexity, Cost

Some API Gateway and ESB Promises

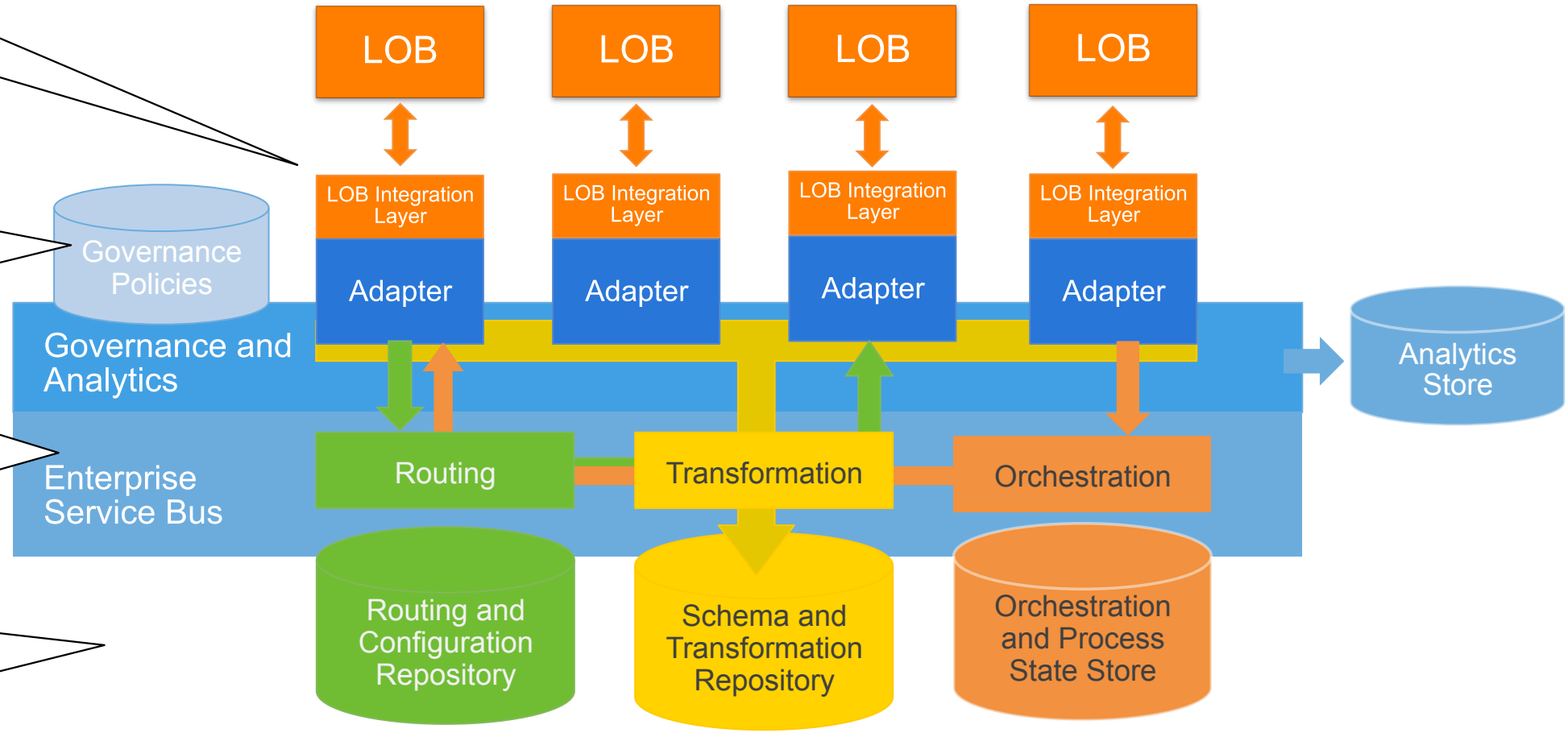
The ESB Model promised simplification through centralization

Centralized Deployment and Config of Integration Logic through Adapters

Centralized Service Governance, Monitoring, and Analytics

Centralized Management of Information Routing, Orchestration, and Transformation

Centralized Enterprise Repositories for Management of Shared Assets



The Centralization Dilemma

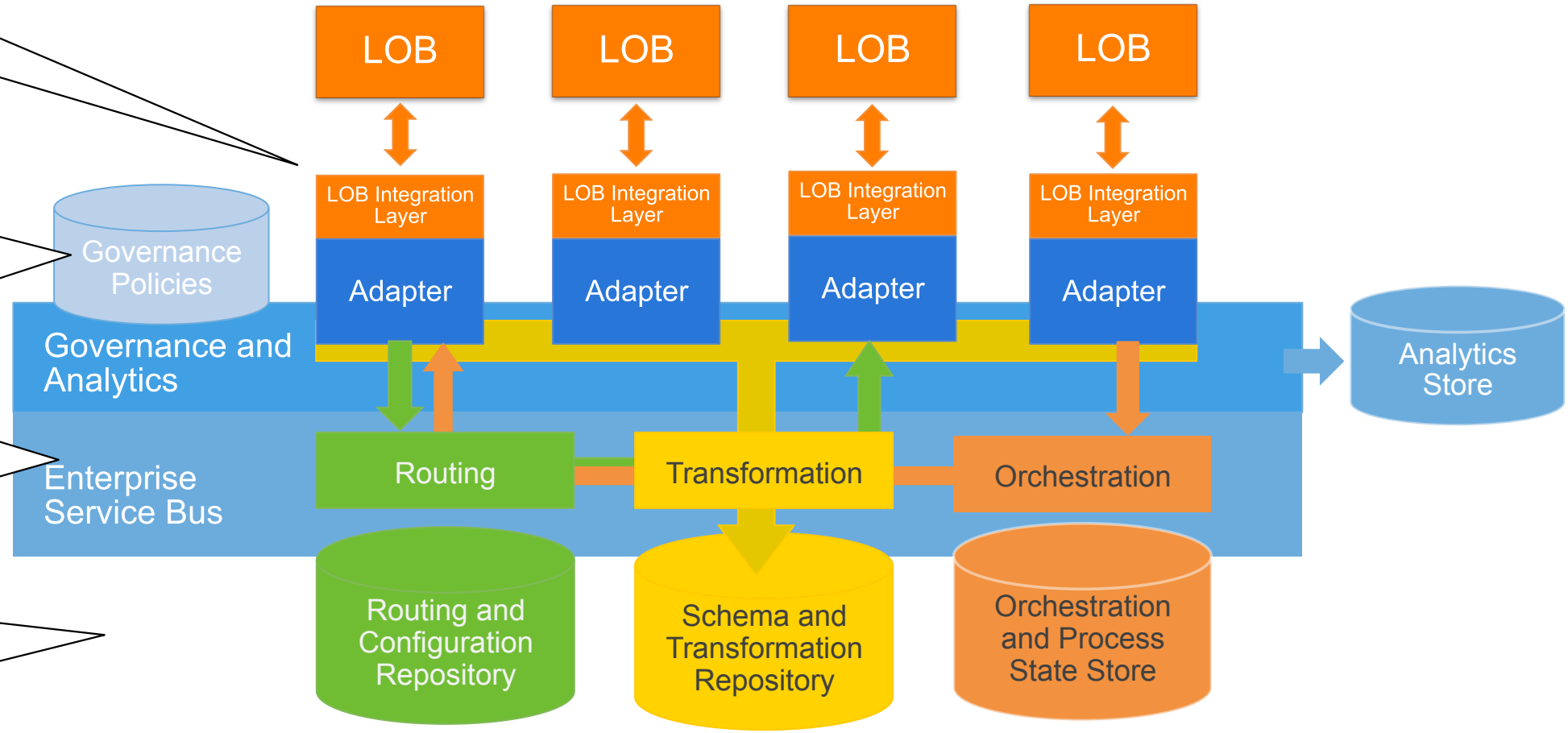
At scale, many ESB/APIG advantages/benefits become weaknesses

Deployment Challenges: Versioning, Distribution, Release Coordination, Platform Dependencies

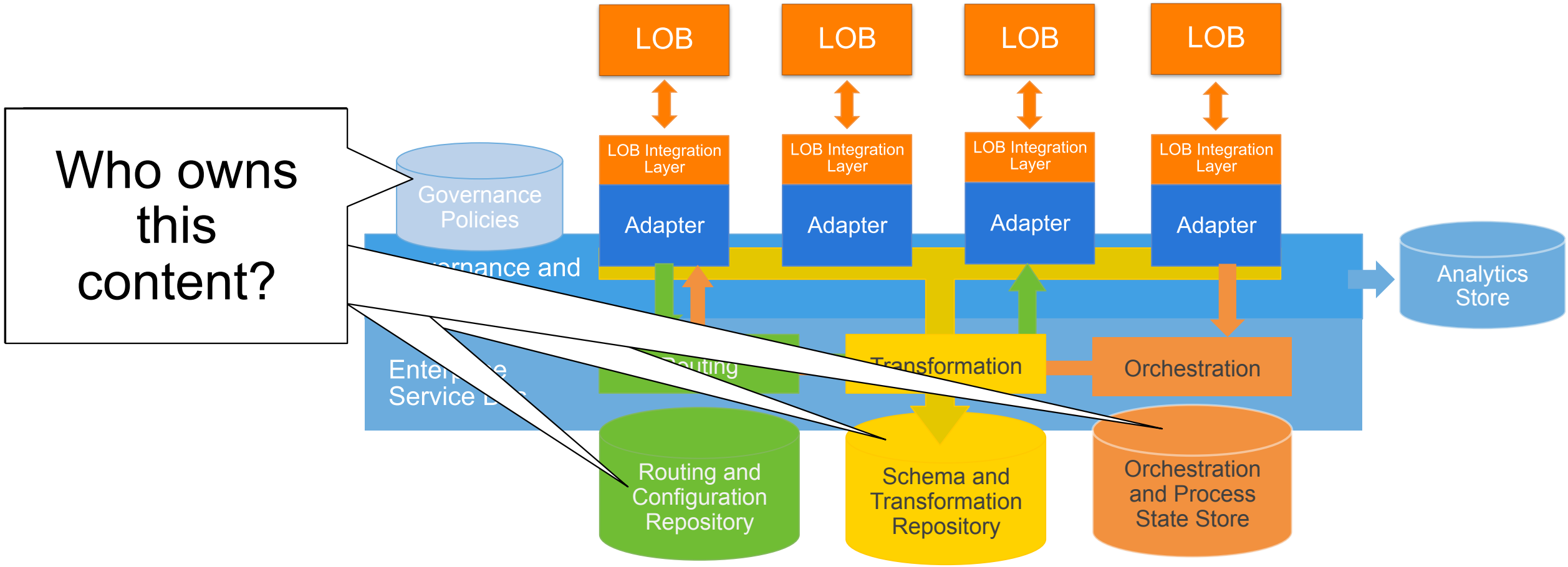
Shared Management Challenges: Config Versioning, Governance Exceptions, Test, Permissions

Scalability Challenges: Scale-Out, Replication, Consistency, Data Volume

Availability Challenges: Geo-Distribution, Disaster Recovery, No Downtime Upgrades



More Centralization Challenges - Ownership



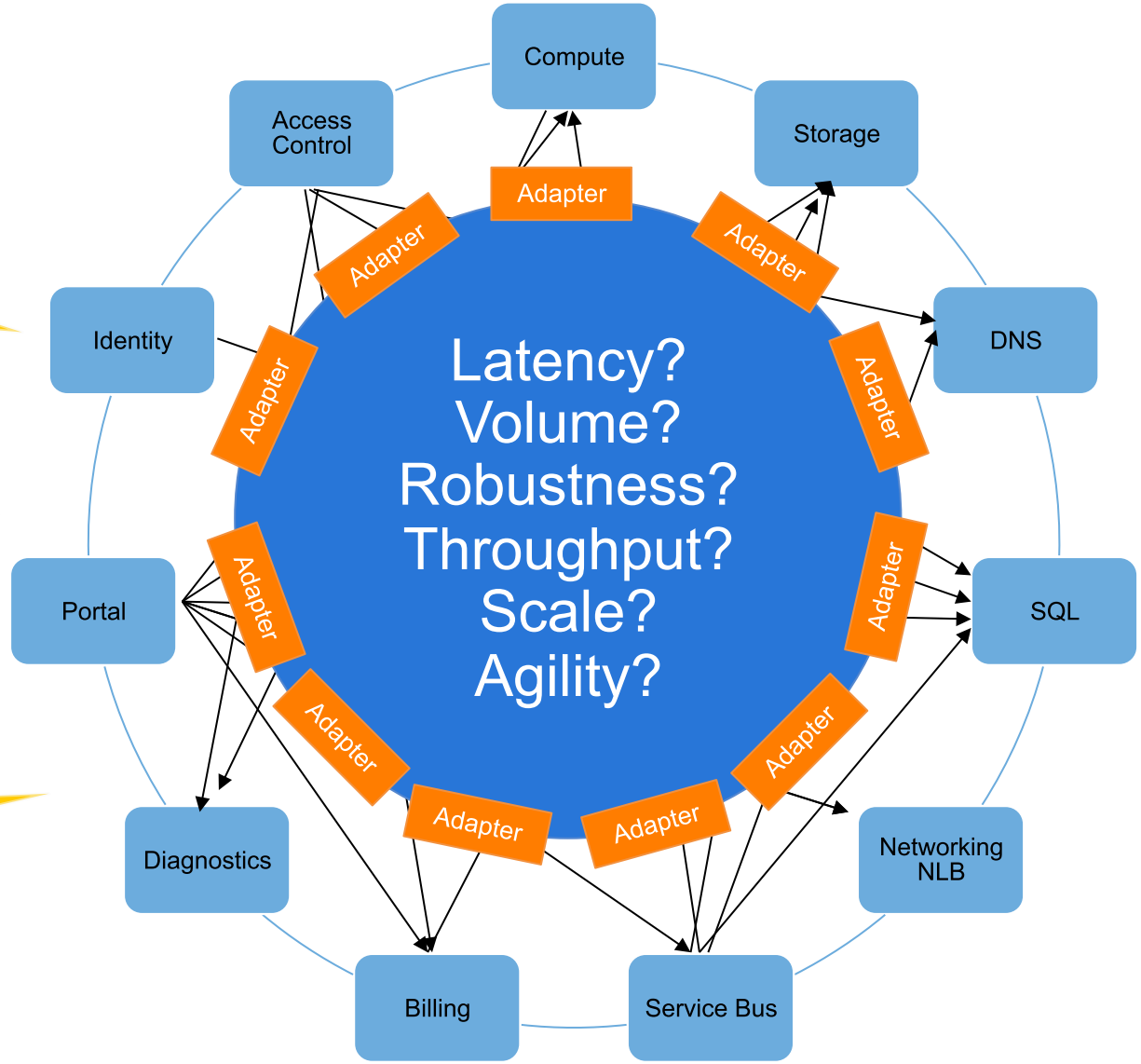
If we'd use the ESB model on Microsoft Azure …

What size and shape would an ESB cluster need to have to function at cloud-scale?

What would the adapters translate from and to?

Are some of these services "special" (SQL? DNS? NLB?) and, if so, why?

Who builds, funds, owns, supports, and operates that central infrastructure?

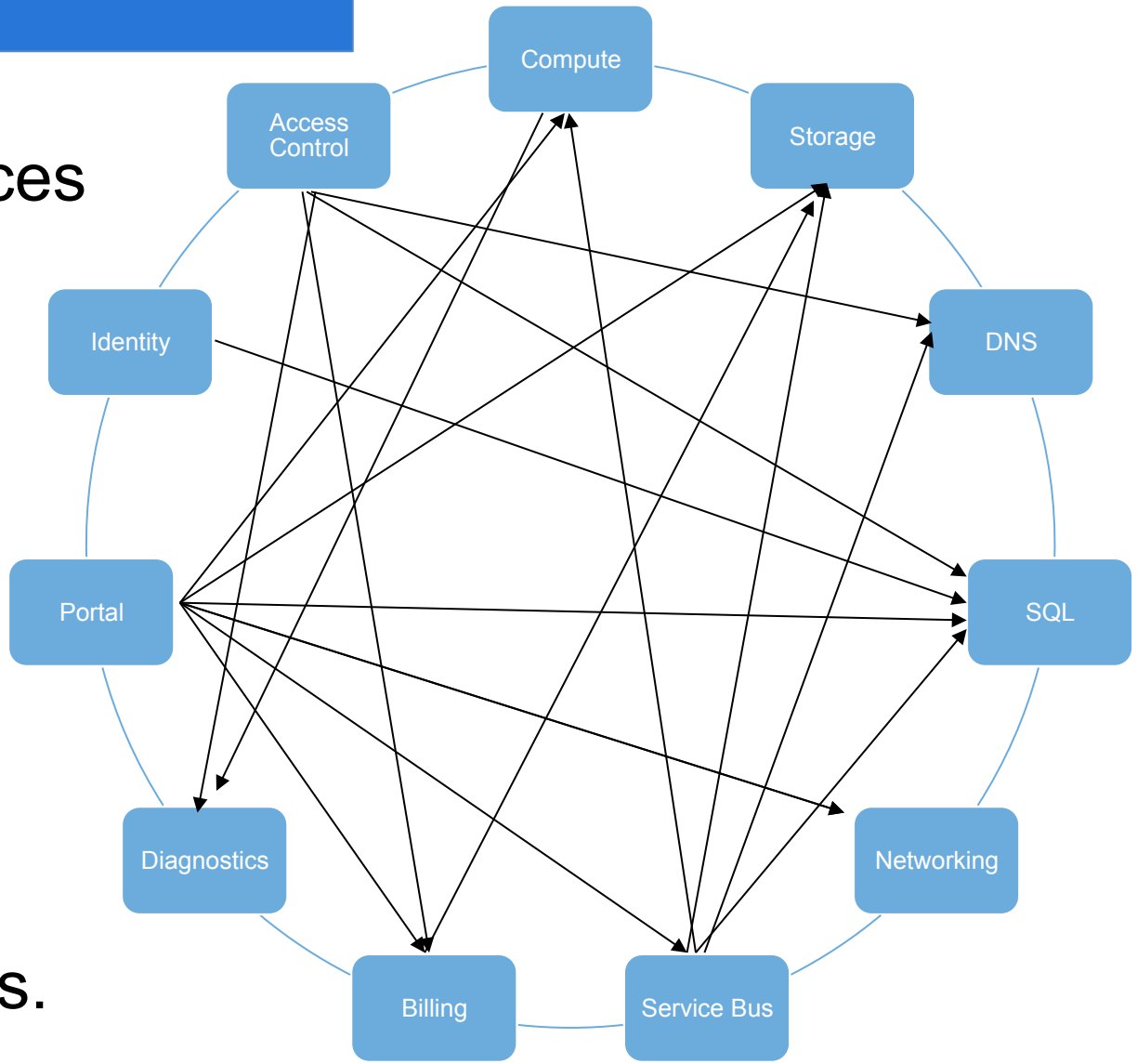


Case-Study Microsoft Azure

- Large Number of Feature Services
- Shared Backend Services
- Independent Teams
- Shipping Features/Fixes Daily
- 28+ Core Datacenter Facilities
- Countless Feature Clusters
- External Partner Integration

- Thousands of very diverse customer workloads

- Federated Autonomous Services.



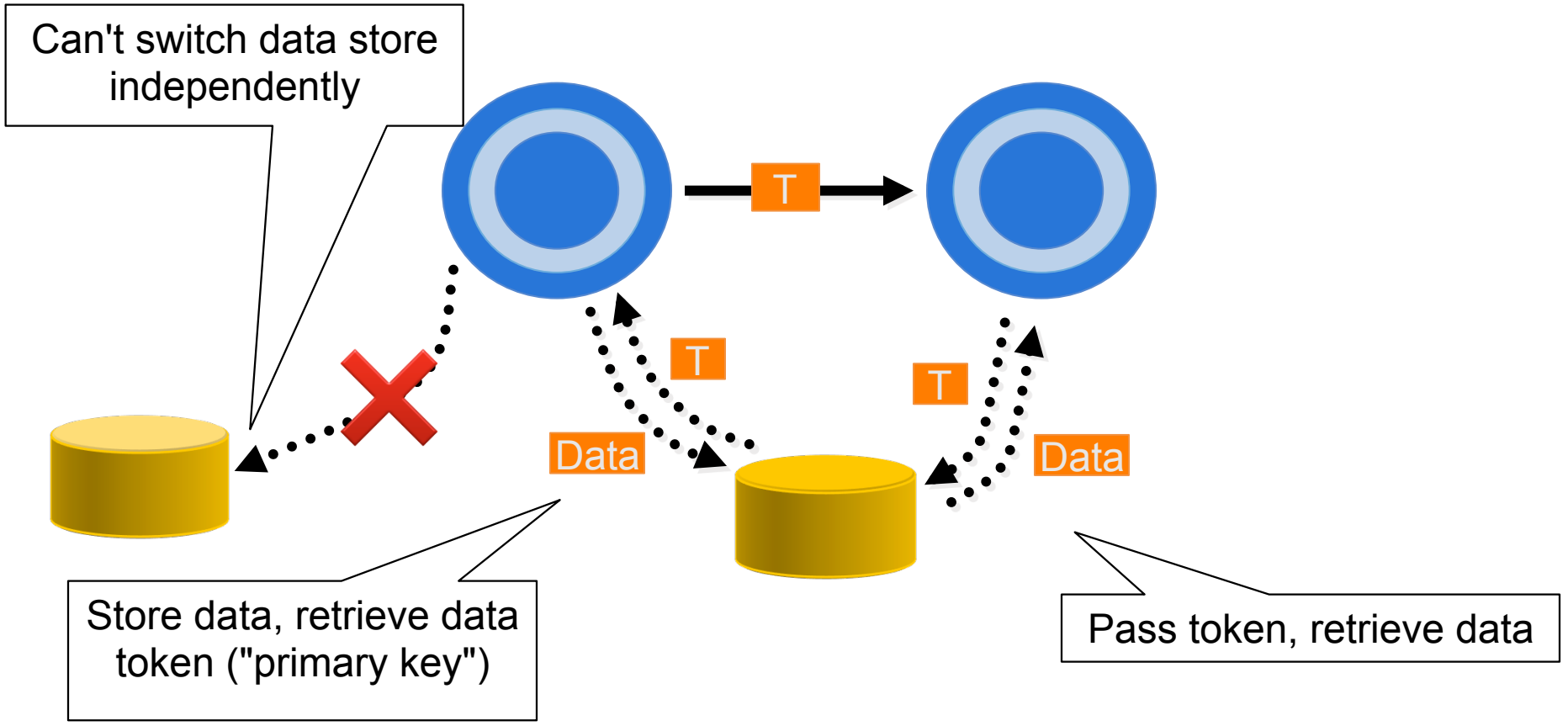
Services: Autonomous Entities

- Defining property of services is that they're *Autonomous*
  - A service owns all of the state it immediately depends on and manages
  - A service owns its communication contract
  - A service can be changed, redeployed, and/or completely replaced
  - A service has a well-known set of communication paths
- Services shall have no shared state with others
  - Don't depend on or assume any common data store
  - Don't depend on any shared in-memory state
- No sideline communications between services
  - No opaque side-effects
  - All communication is explicit
- **Autonomy is about agility and cross-org collaboration**

Interdependencies

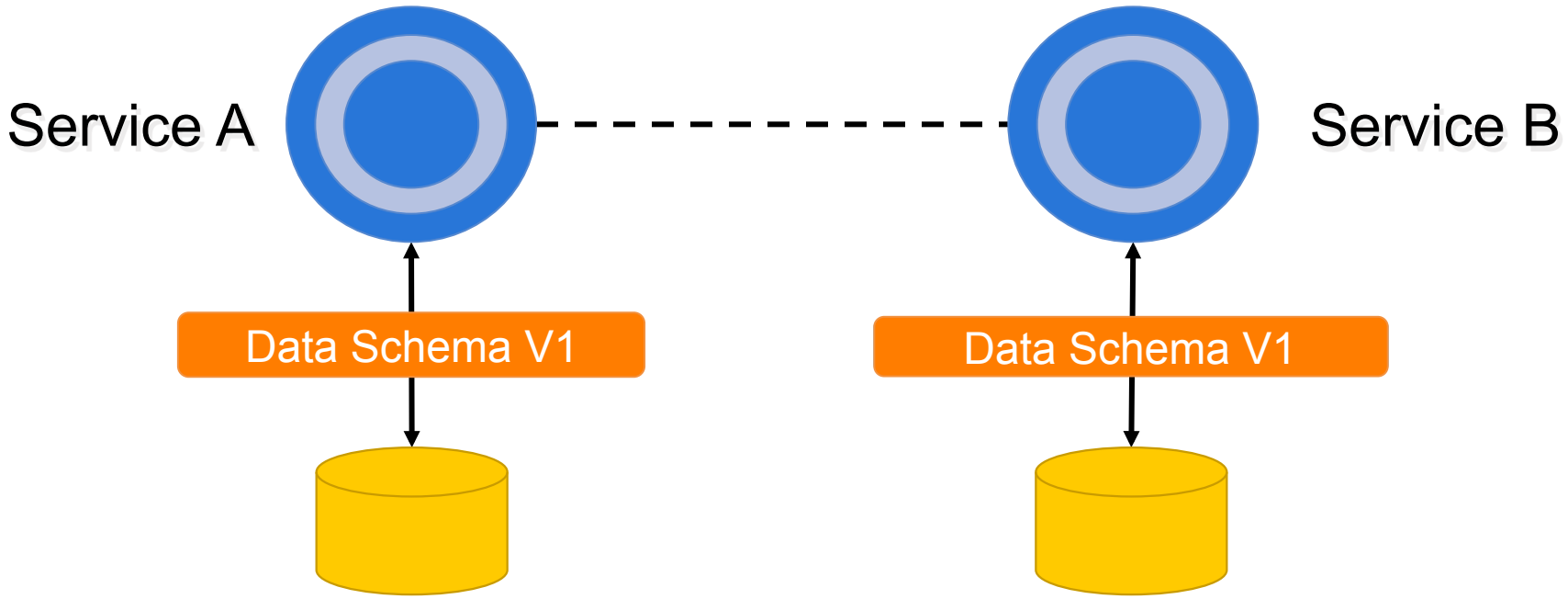
- An autonomous service owns its own uptime
  - If a downstream dependency service is unavailable, it may be acceptable to partially degrade capability, but it's not acceptable to go down blaming others
  - Any critical downstream dependencies need to be highly available, with provisions for disaster recovery.
  - A service can rely on a highly-available messaging middleware layer as a gateway to allow for variable load or servicing needs
- An autonomous service honors its contract
  - Version N honors the contract of Version N-1. Contracts are assurances.
  - Deprecation of a contract breaks dependents; have a clear policy

Why Shared Data Stores Are Bad

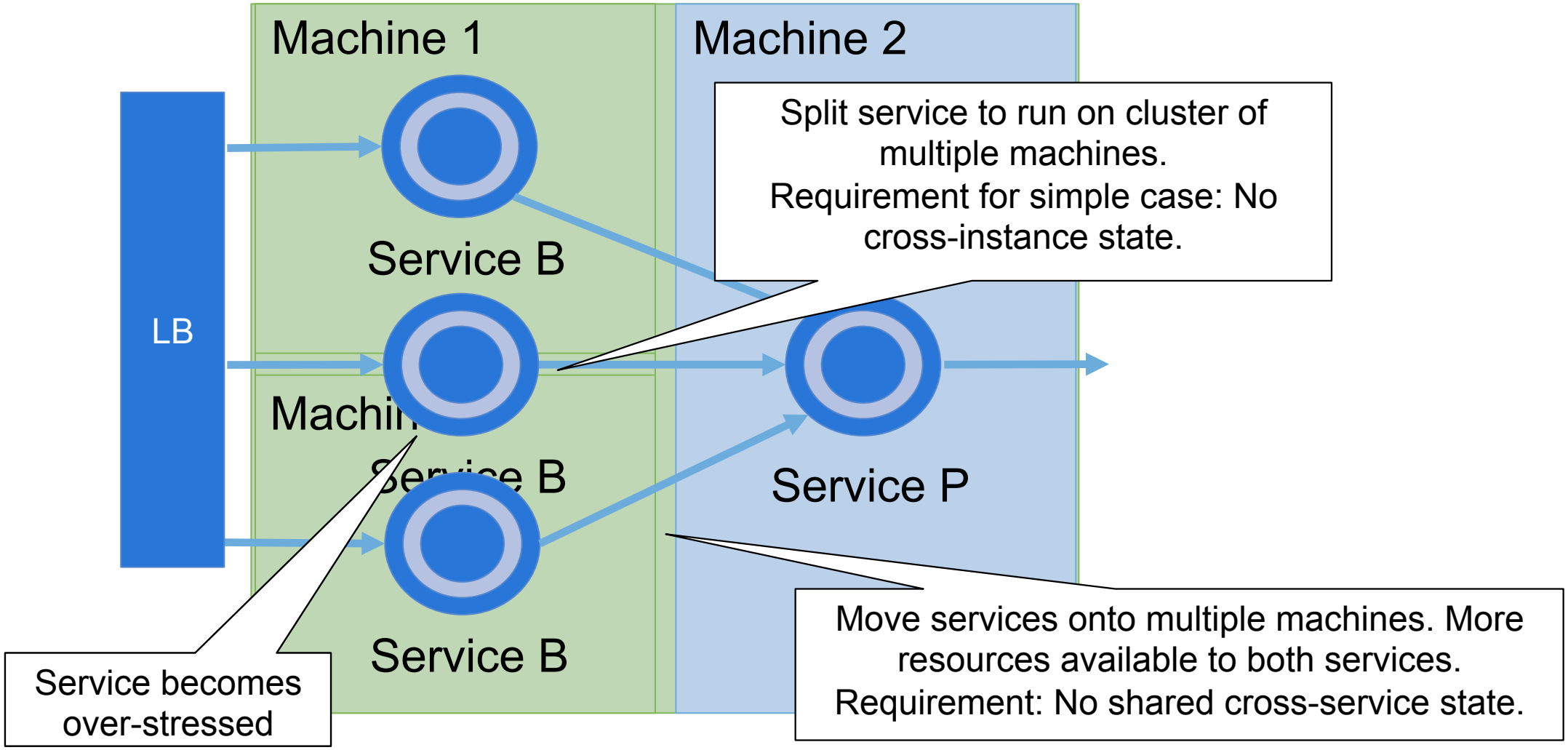


Assumption about shared data store

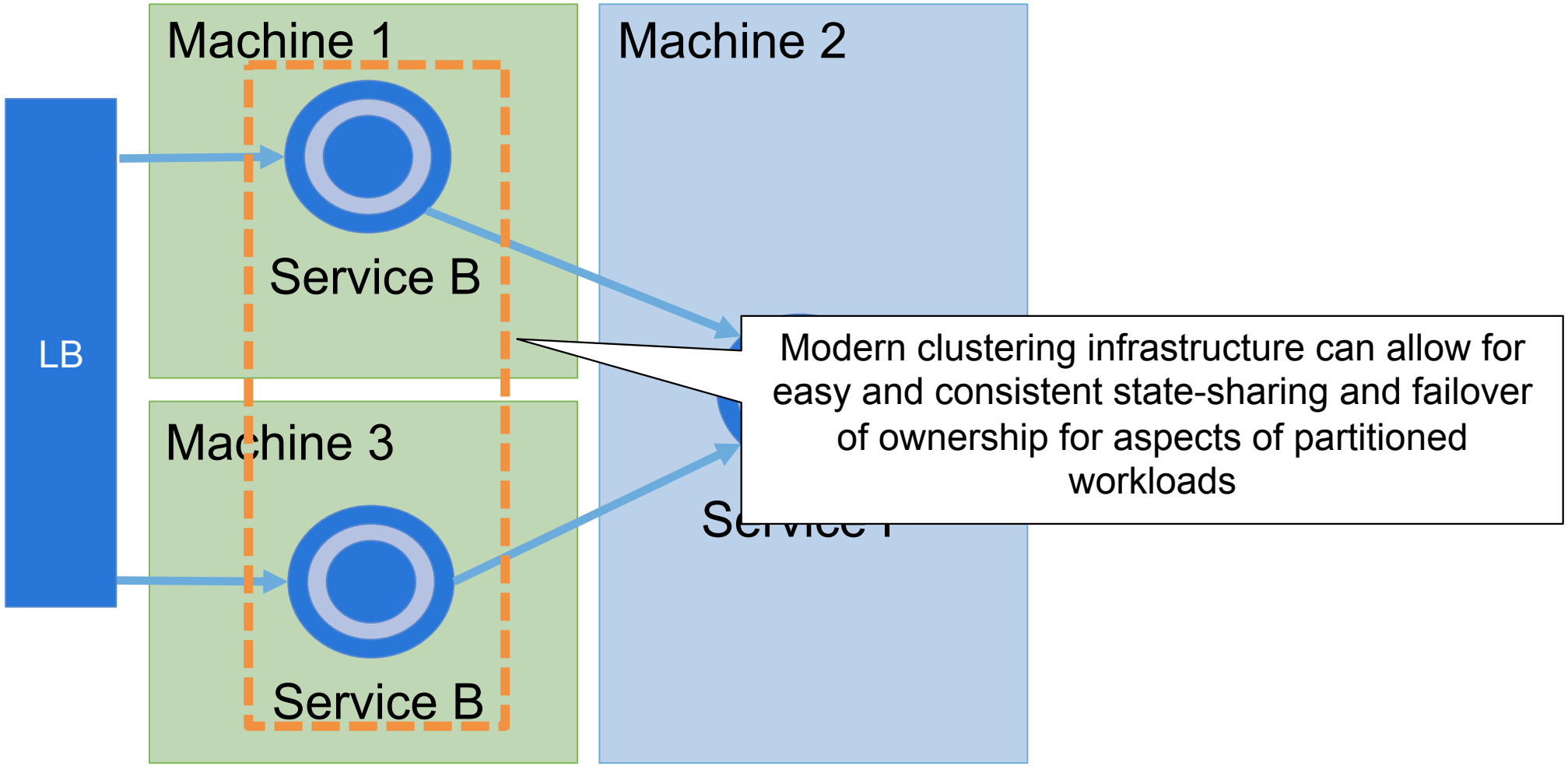
Data Store Decoupling Enables Evolution



Clustering


Machine 1
Machine 2
LB
Service B
Service B
Service B
Service P
Split service to run on cluster of multiple machines.
Requirement for simple case: No cross-instance state.
Service becomes over-stressed
Move services onto multiple machines. More resources available to both services.
Requirement: No shared cross-service state.

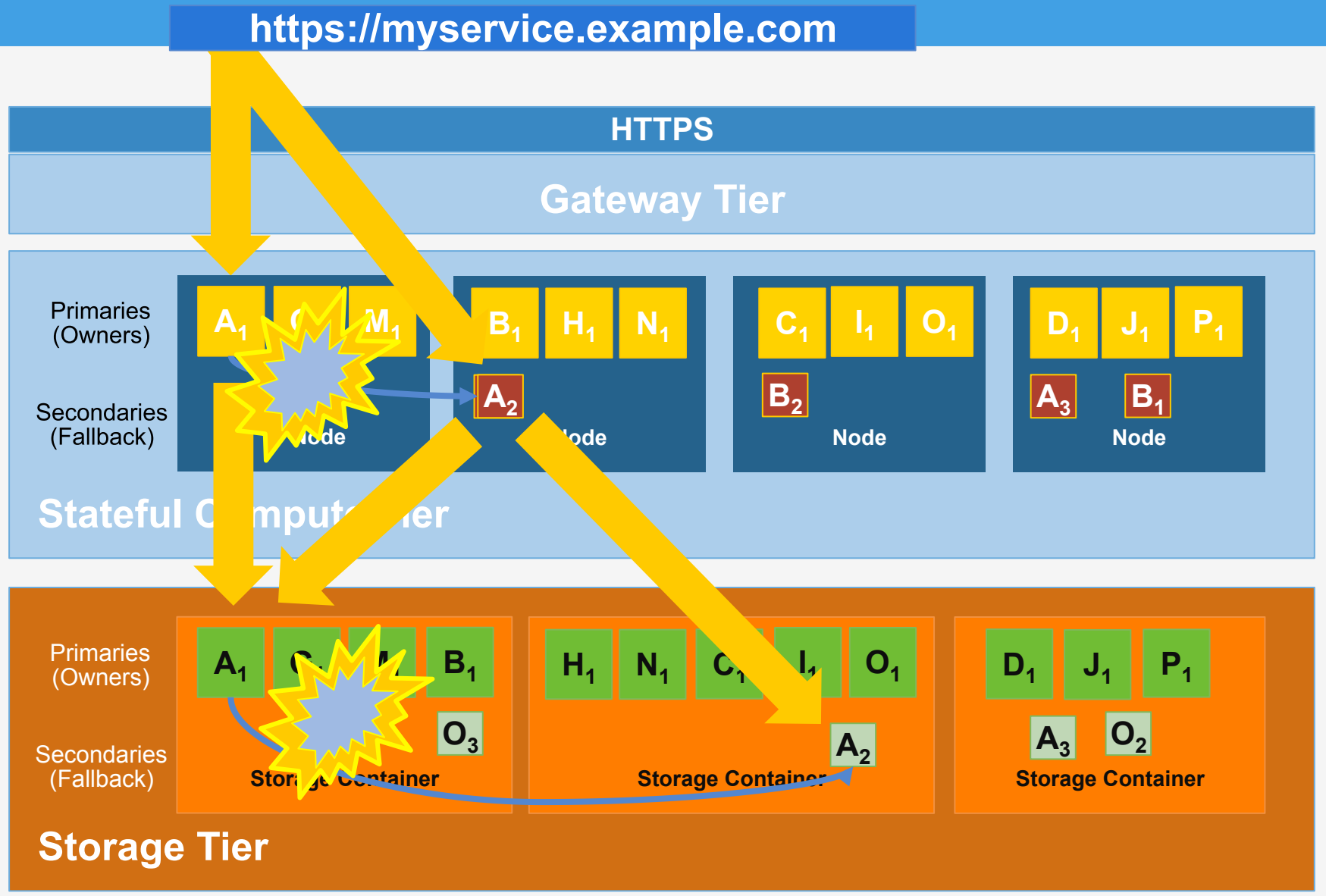
Clustering


Machine 1
Machine 2
LB
Service B
Machine 3
Service B
Modern clustering infrastructure can allow for easy and consistent state-sharing and failover of ownership for aspects of partitioned workloads

Multi-Node Failover Clustering

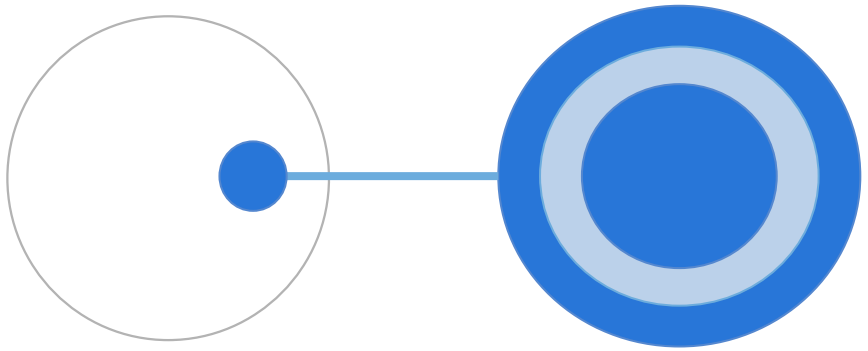
Failure of any node – in gateway, compute, or storage – leads to an automated "failover" to one of at least two secondaries.

Secondaries are continuously updated with the all information required to instantly take ownership when needed.



“Fiefdoms and Emissaries”

- Term coined ~2002 by @PatHelland
- “Fiefdom”: Autonomous Service
- “Emissary”: Logic/Code
  - JavaScript on Web Pages
  - Client SDKs
- „A service owns its contract“ can also manifest in it owning SDKs for all relevant platforms while keeping the wire contract private.
- We‘ll see more of this around “edge compute“ and “ fog“ in IoT



Autonomous Services Benefits

- Autonomy enables operational agility
  - Services and all of their implementation elements can be moved, reconfigured, and replaced “behind the curtain”
- Autonomy enables clustering
  - Scalability: Can add more capacity and introduce partitioning/sharding
  - Reliability: Can transparently compensate for failures
- Autonomy enables reusability and adaptability
  - Services can be used from other contexts
  - Services can be replaced if contract carries forward

Operational Objectives

- Scalability
  - Scale to requirements. Up (workload growth), down (resource constraints), out (more resources)
- Availability
  - Keep the system available as required for the solution.
- Consistency
  - Provide a view of the information held by a system that is as consistent as needed to fulfill the solution requirements

Operational Objectives

- Reliability
  - Operate the system reliably and resilient against failures
- Predictability
  - Design to achieve predictable system performance
- Security
  - Identify and explicitly mitigate (or choose not to mitigate) security threats.

Operational Objectives

- Agility
  - Design the system such that defects can be corrected and new capabilities introduced while meeting operational objectives.
- Safety
  - Provide safety for data and systems by mitigating the risk of disasters impacting the existing environment(s).
- Supportability
  - Create systems to provide operational transparency for the needs of operations and support staff
- Cost
  - Do all of the above within a set budget and striving to continually reduce that cost.

Operational Assurances

- Service owners aim to meet operational objectives so that they can provide operational assurances:
- What level objective achievement can and does the service owner commercially commit to?
  - Example: Operational objective 99.99% availability/week (10 minutes max downtime) might turn into assurance 99.95% (50 minutes max downtime)
  - Latency? Throughput? Data Loss? Disaster/Failure Recovery Time?
- What is the support lifecycle commitment for APIs and contracts?
  - How many versions? Minimum deprecation notice?

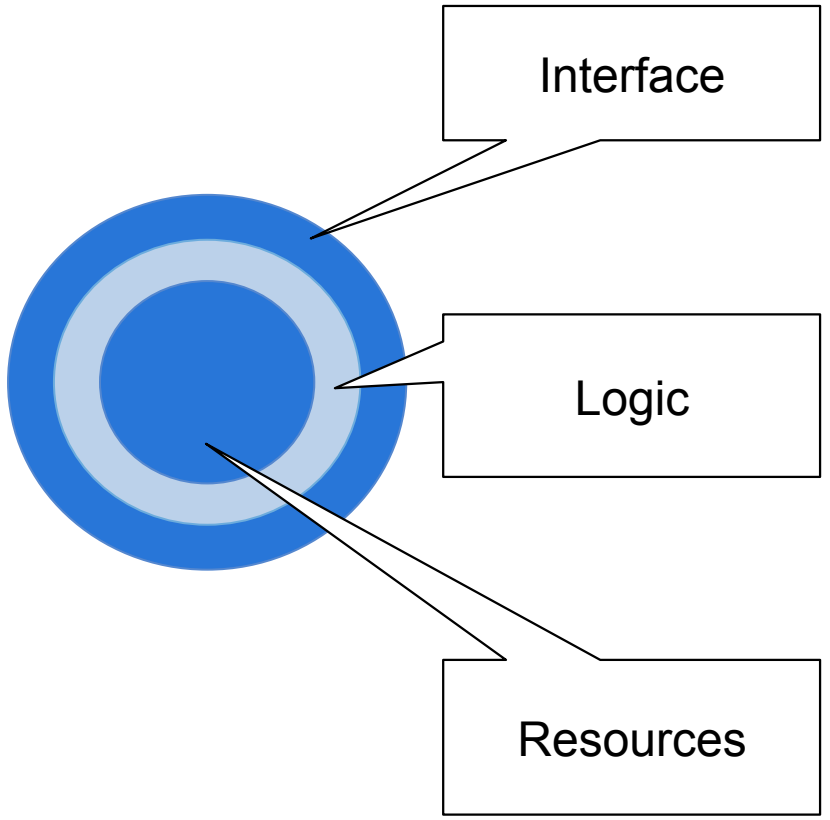
Layers, Tiers, and Services

The implementation of a service is often organized into functional layers, and those layers may span multiple tiers

Layers: Code Organization

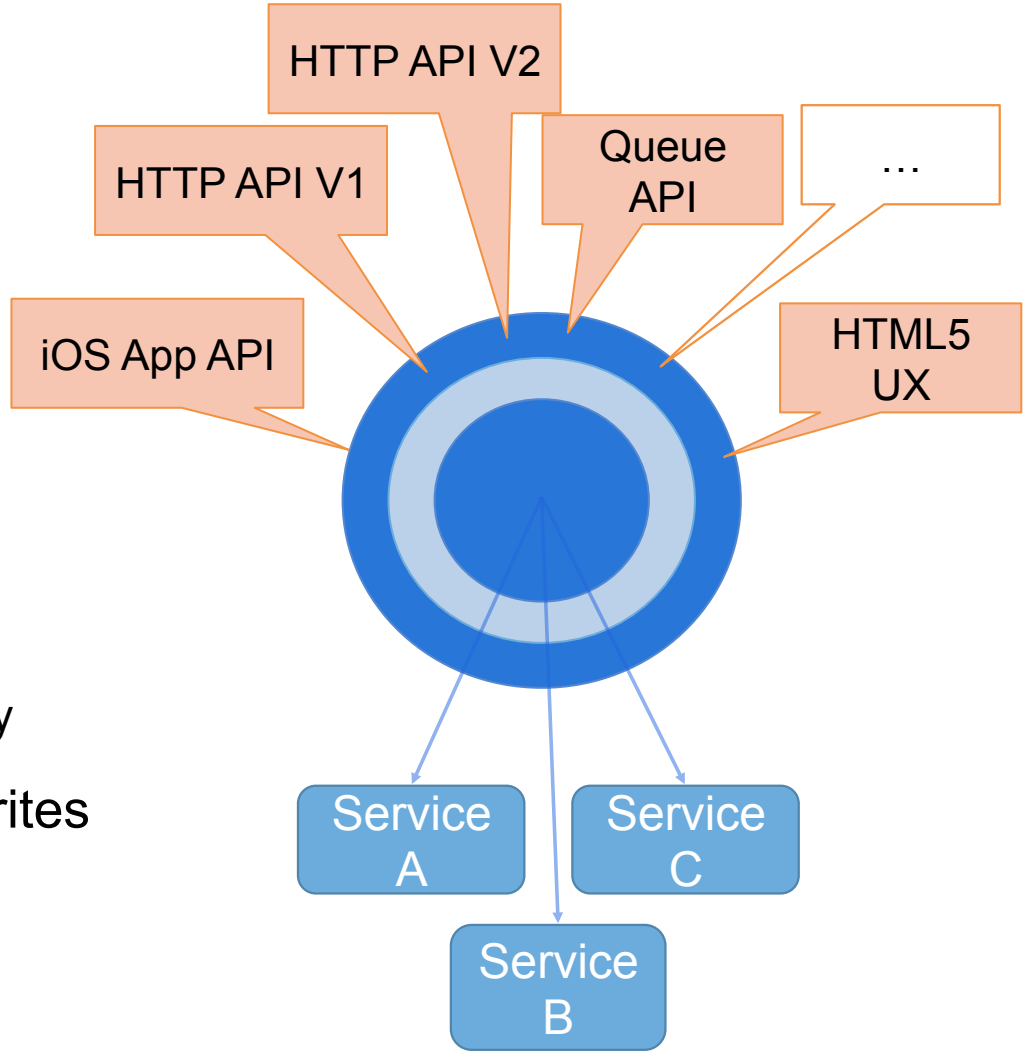
- We usually structure implementation (code) into several distinct layers. Most commonly:

- “Interface” captures information
  - Presentation Events
    - HTML, GUI, Web Services, Pipes, Queues, RPC, …
  - System Events
    - Timers, OS Wait Objects, Alerts, …
- “Logic” filters, validates, and processes information
  - Functions, Classes, Lambdas, Actors, etc.
- “Resources” are platform
  - Web Services, Databases, Queues, …



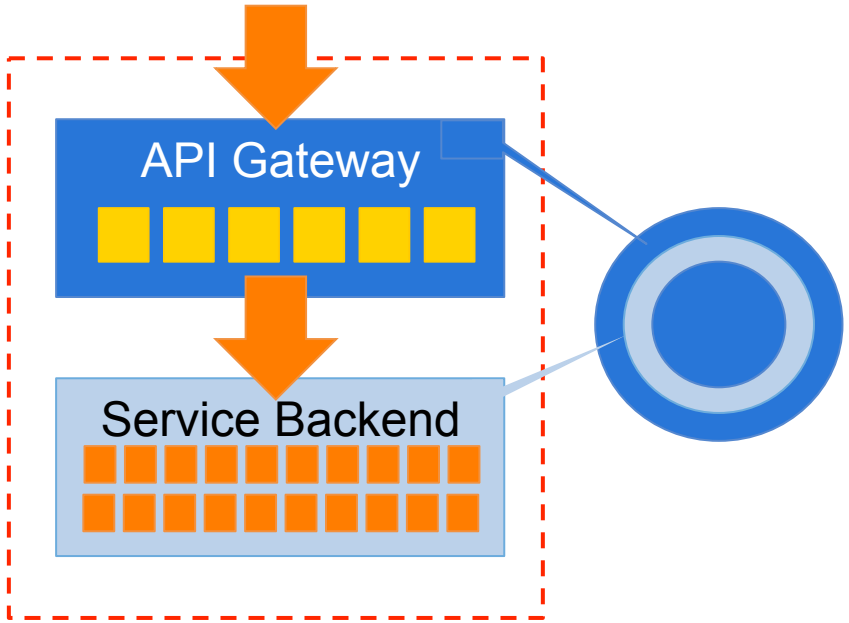
Rationale for Layers

- Key rationale for layers: Resilience against changes in ambient contracts.
- Communication and Presentation Layers
  - Lots of changes, fairly frequently
    - New UX methods and layouts, new assets
    - New contracts and schemas
    - New protocols
  - Can have multiple concurrent interfaces
  - Each change has low impact, but work adds up
- Resource Access Layers
  - Fewer changes, rather infrequently
    - Downstream dependency services make compatibility assurances
  - Sometimes massive impact, often wholesale rewrites
- Goal is for core logic to be resilient against interface changes

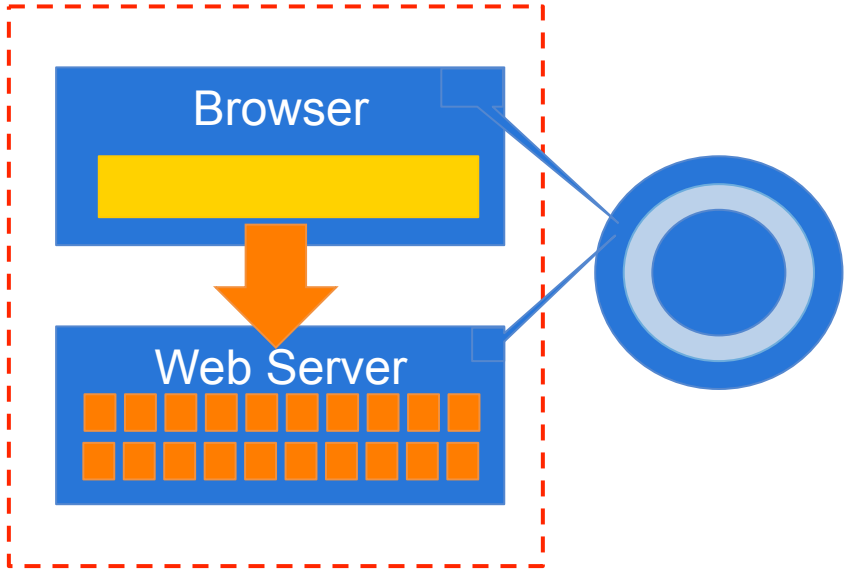


Tiers: Runtime Organization

- Tiers are about meeting operational objectives
  - Aspects of one service or even one layer may have different scalability and reliability goals
  - Resource governance (I/O, CPU, Memory) needs may differ between particular functions
  - UX tier will be more efficient and more adaptable with client-based rendering
- Tier boundary most often is a process boundary
  - On same machine, across machines
  - In same organization, across organizations
  - In trusted environment, across trust boundaries
- Tier boundaries often cut through layers
  - Cuts may separate “yours” and “theirs”
  - Ex: “Your” hosted web code and “their” browser
  - Ex: “Your” data access code and “their” database

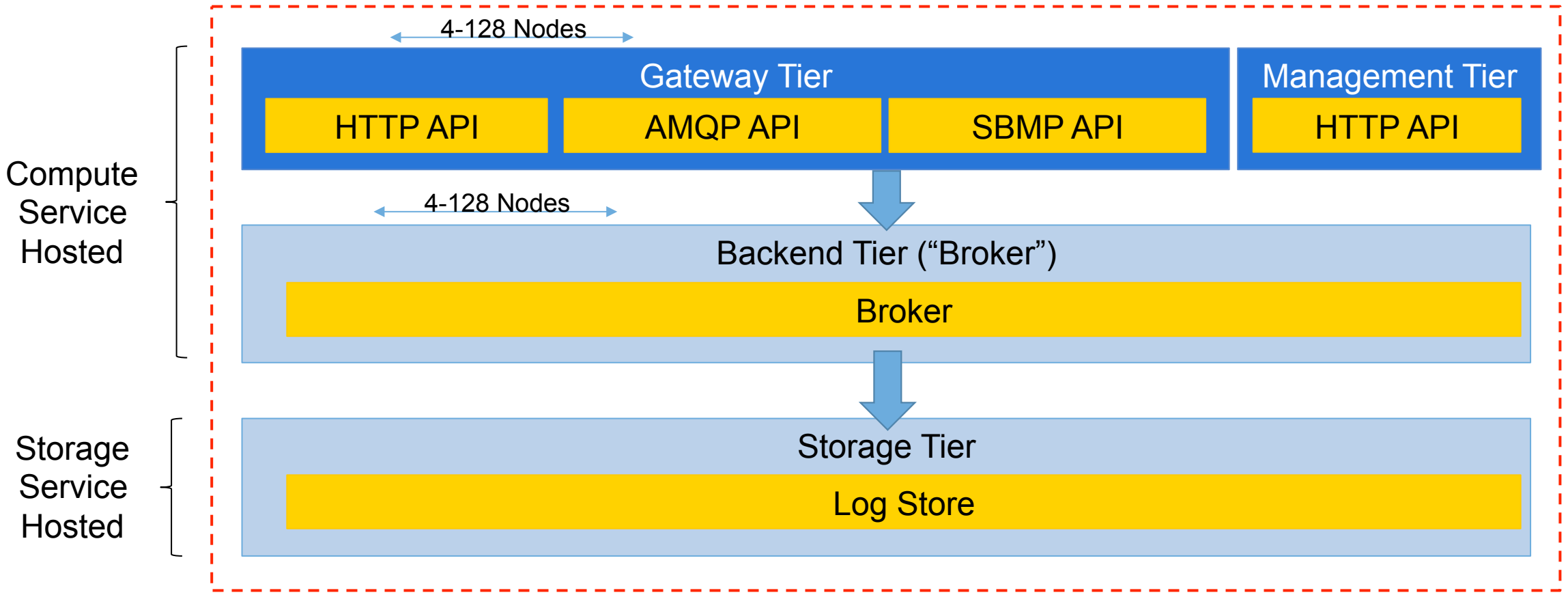


2 tiers, 2 layers



2 tiers, 1 layer

Example: Azure Service Bus



Layers, Tiers, and Services

- Layers:
  Code Management
- Tiers:
  Runtime Management
- Services:
  Ownership Management

The implementation of a service consists of one or multiple deployment tiers that implement one or multiple layers

A “service” is a software and operations deliverable owned by an autonomous organization.

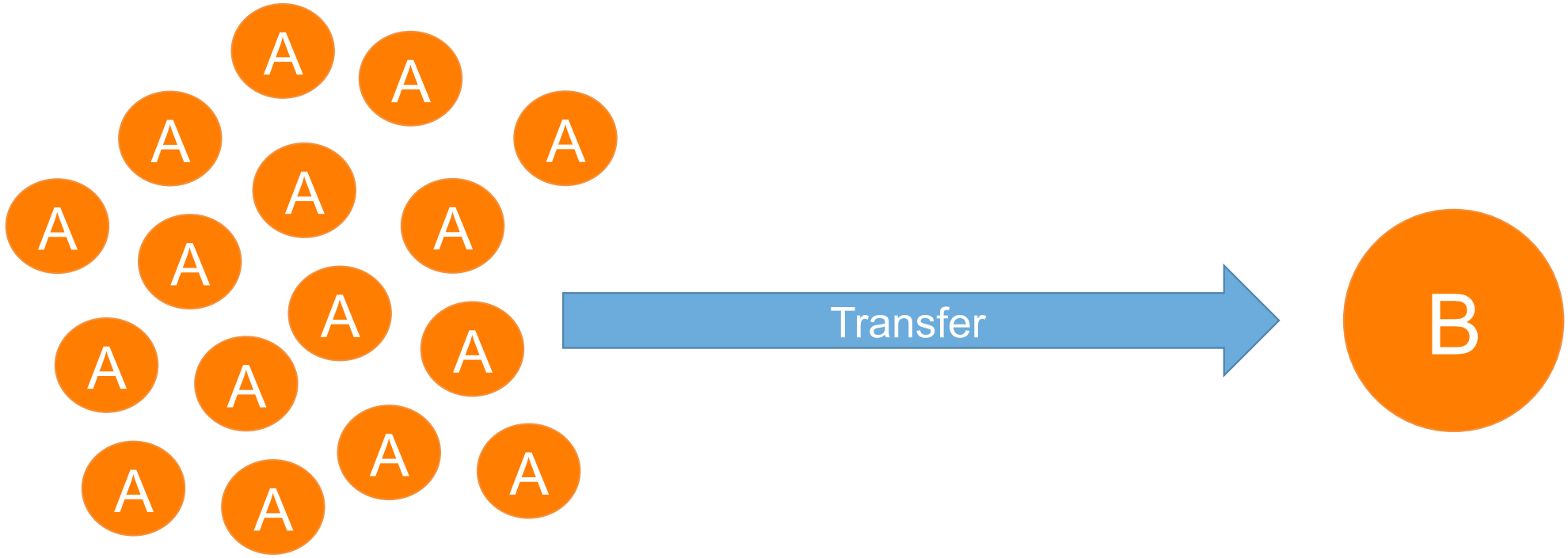
Communication

How do we move information between systems?

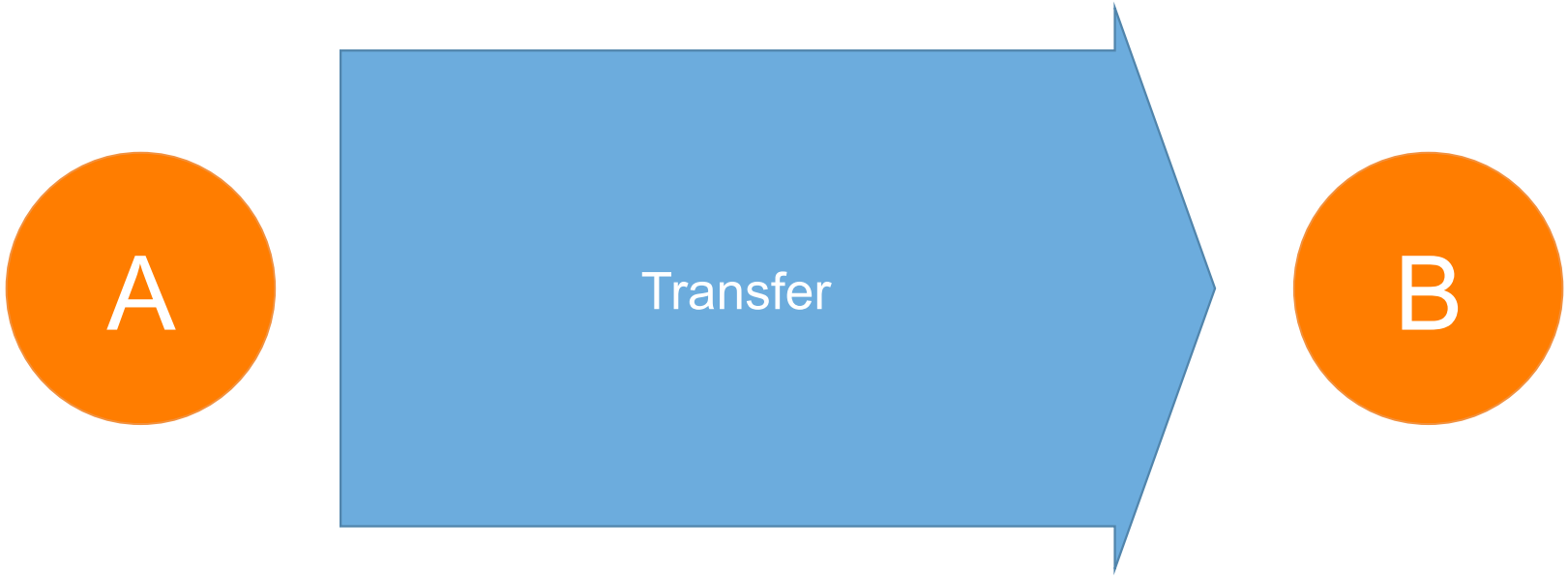
Direct	Brokered	Broadcast
Moving events between two endpoints.	Moving events via communication middleware	Distribution of events from one source to anyone interested
“Phone”	“Mail”	“Radio”

Messaging is all about getting data from here to there

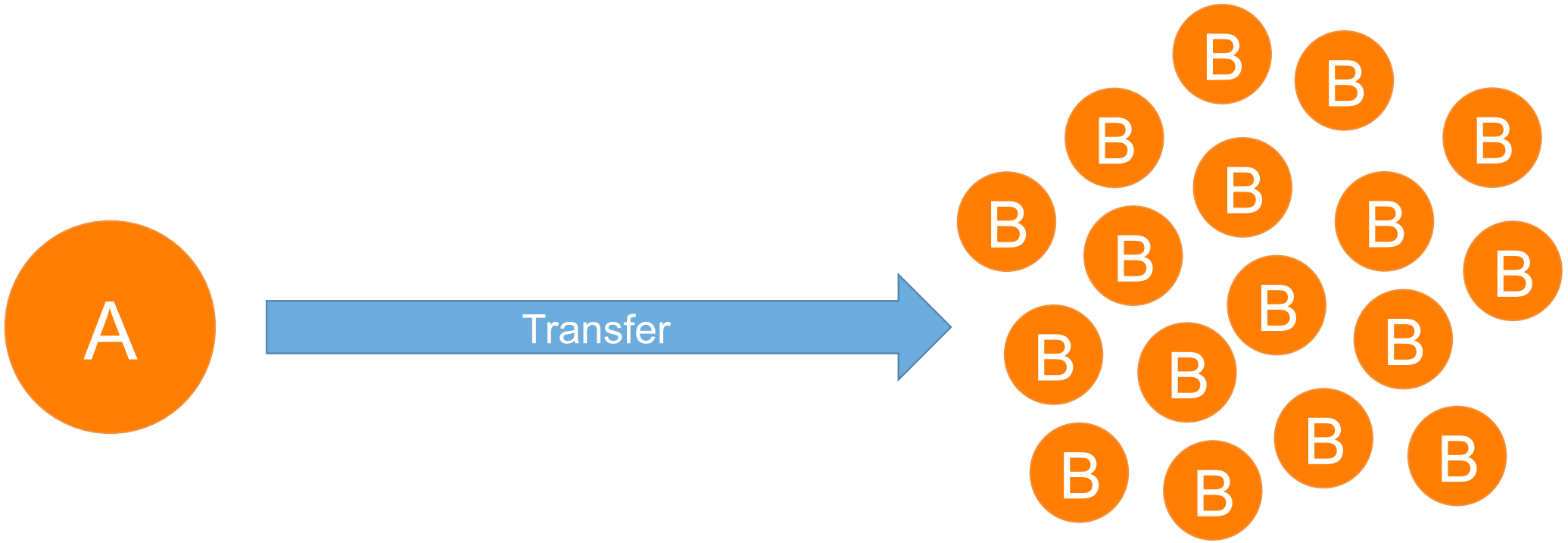
(Getting data back from there to here is the just same thing)



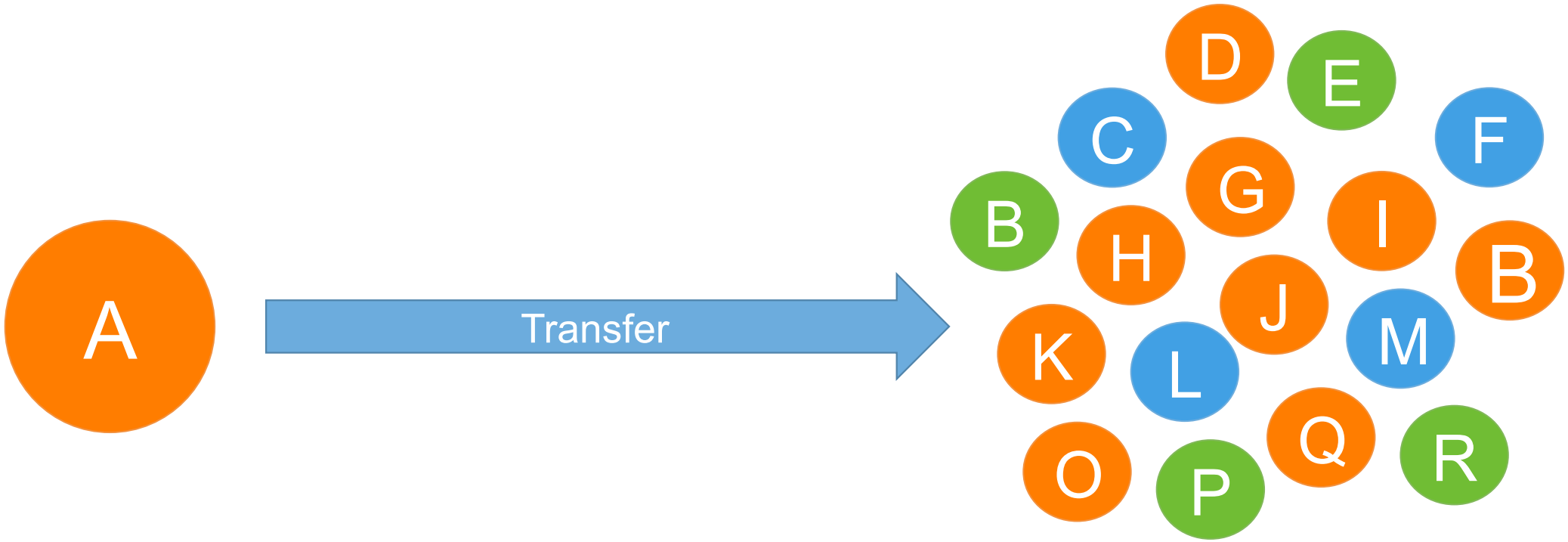
Sometimes there’s a lot of “here”



Sometimes there’s A LOT of data



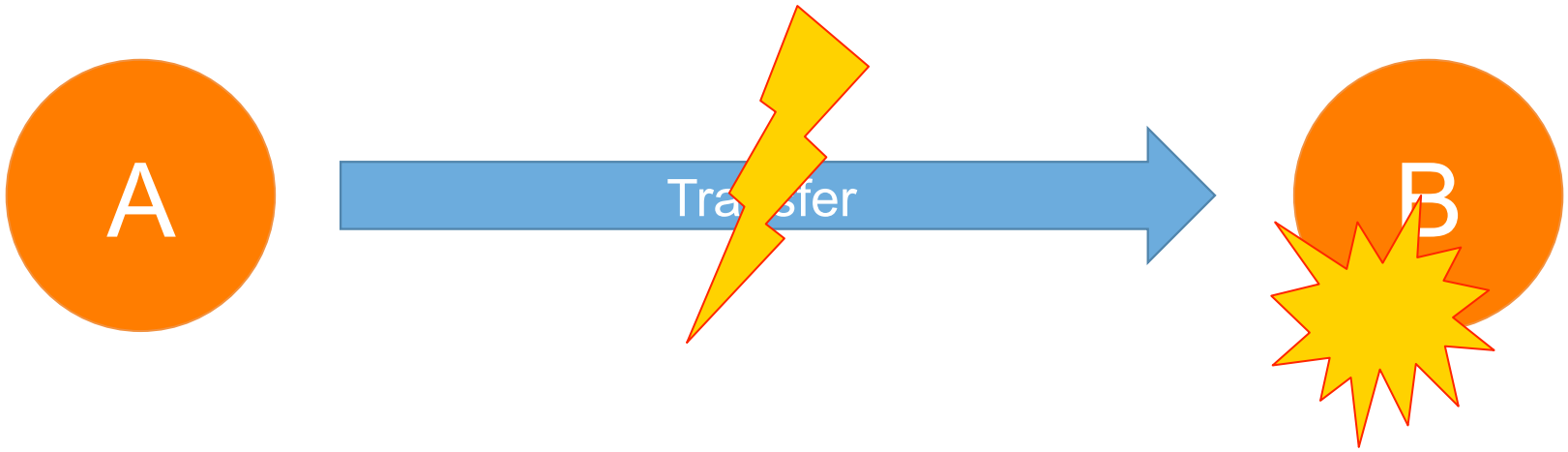
Sometimes there’s a lot of “there”



Sometimes the “there” are all different

Sometimes “there” isn’t currently paying attention

Sometimes “there” is VERY BUSY



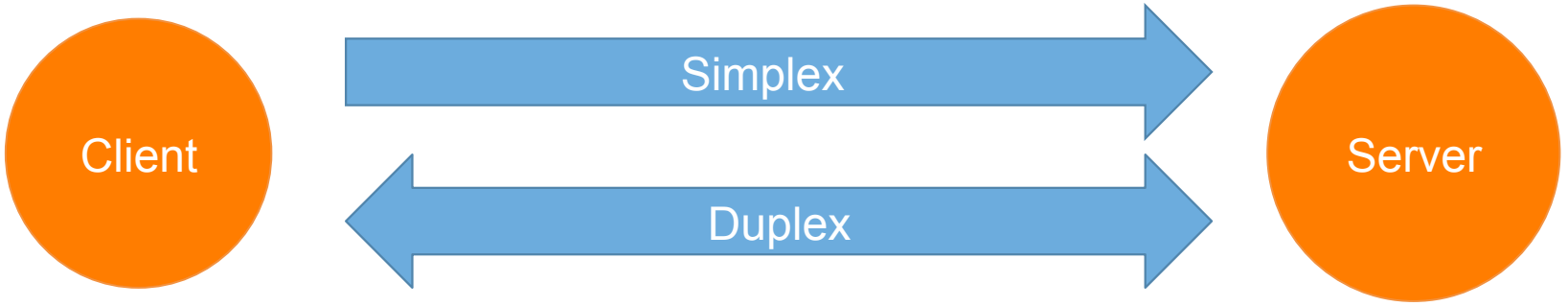
Sometimes there’s trouble

Client vs. Server

- A "client" commonly decides which "server" it wants to talk to and when.
- The client needs to locate the server, choose a protocol the server provides, and initiate a connection.
- The client will then typically provide some form of authentication proof as part of the connection handshake

- A "server" commonly listens for client-initiated connections, on one or multiple network protocol endpoints.
- Once a client attempts to connect, the server will typically request some authentication proof that is then validated for access authorization.
- The server needs to deal with any malformed or malicious requests

Directionality



- A **simplex** (or uni-directional) protocol allows flow of data in just one direction.
- A **duplex** (or bi-directional) protocol allows independent flow of data in both directions.
  - Half-duplex only allows one of the parties to communicate at a time
  - Full-duplex allows both parties to communicate concurrently

Symmetry

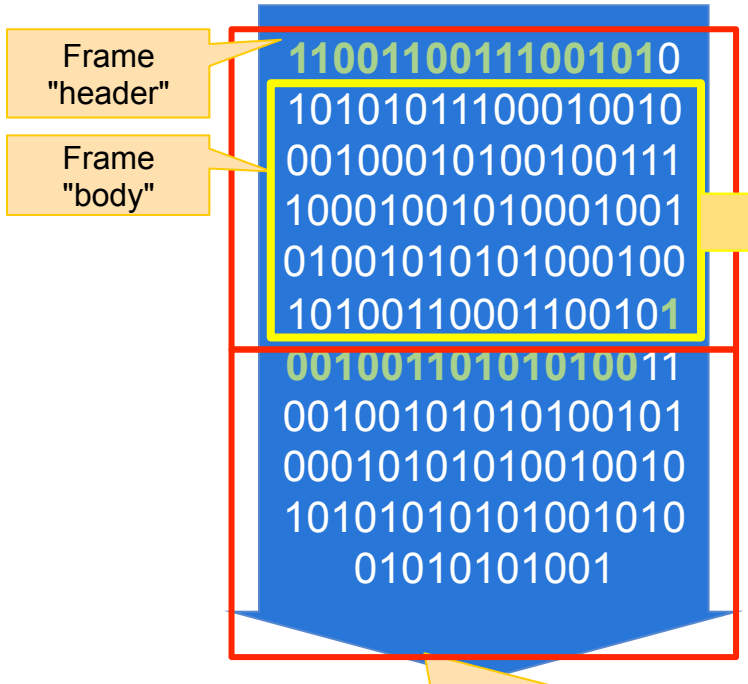
- A protocol is **symmetric** when is allows all of its supported gestures (except for connection establishment) independent of who initiated the connection.

Multiplexing

- **Multiplexing** allows a singular network connection to be used for multiple concurrent communication sessions (or links)
- Establishing connections can be enormously costly, multiplexing saves the effort for further connections between parties

Framing, Encoding, Data Layout

Framing



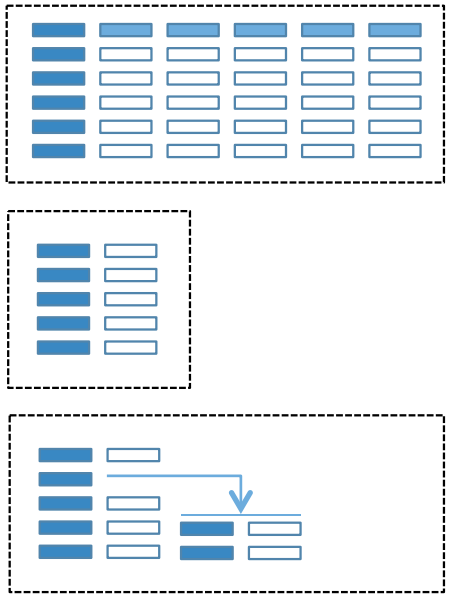
A message or framing protocol splits the data stream into distinct chunks that can be processed in sequence or separately.
A frame/message header contains information about the size, content, destination, and often also an expression of the the sender's intent

Encoding

JSON	Avro	Msg Pack
XML	AMQ P	CSV
MPE G	Text	Raw

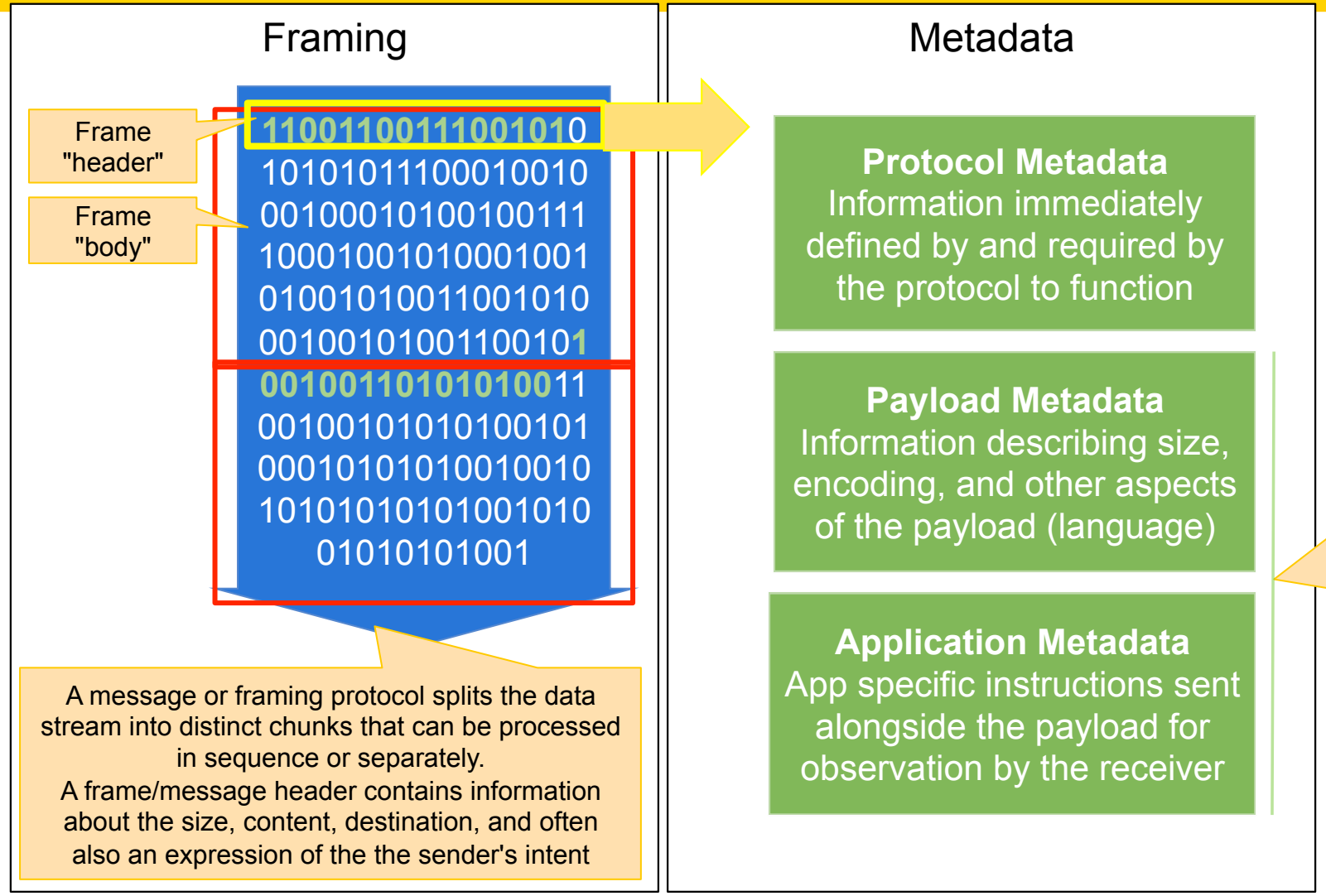
The encoding (or media-type) tells a message processor how to interpret the payload data of the message.

Layout



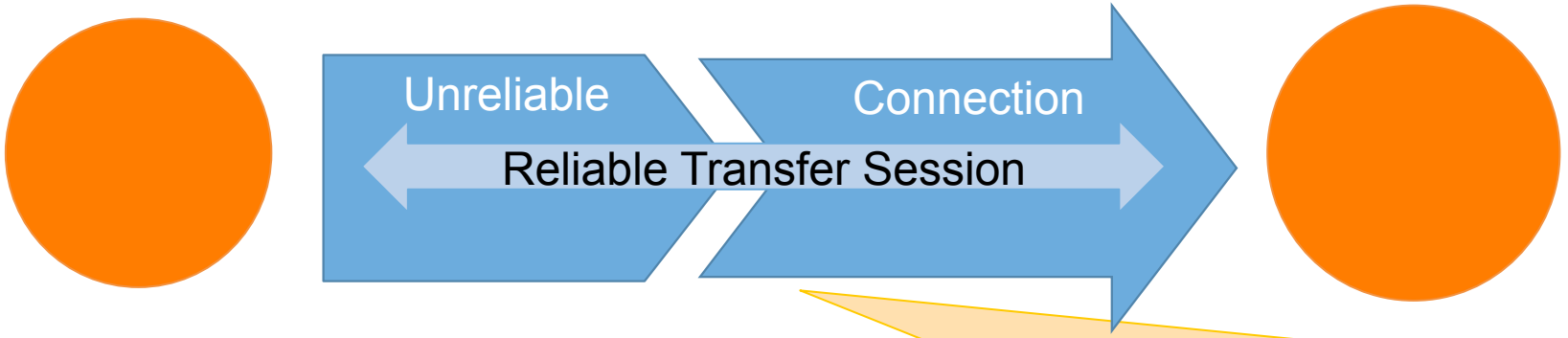
A data layout convention can tell a processor how to deal with structured data in a dynamic fashion to distinguish objects or rows/columns

Metadata



- Not all protocols allow for payload and application metadata, requiring externally agreed conventions establishing mutual understanding of message content

Transfer Assurances



- Reliable protocols allow transfer of frames more reliably than underlying protocol layers
  - Compensating for data loss, preventing duplication, ensuring order
- Various strategies to compensate for data loss
  - Resend on negative acknowledgment („data didn‘t get here“)
  - Resend on absence of acknowledgment
  - Send duplicates of frames
- **Common Transfer Assurances**
  - "Best Effort" or "At Most Once" – no resend, not reliable
  - "At Least Once" – frame is resent until it is understood that is has been delivered at least once
  - "Exactly Once" – frame is delivered exactly once [see next]

The Edge of Services

- In this context, a service is:
  - A reusable artifact, that can be accessed through channels, well-defined by some interface contract
  - These communication protocols stress interoperability and location transparency – just more or less
- A single communication mechanism does not fit all uses!
  - The very same interface may be reachable by several channels
  - The service may be located on the same machine or on the other side of the world
- How can I reach out to a service?
  - I.e. what kind of channels can I use?

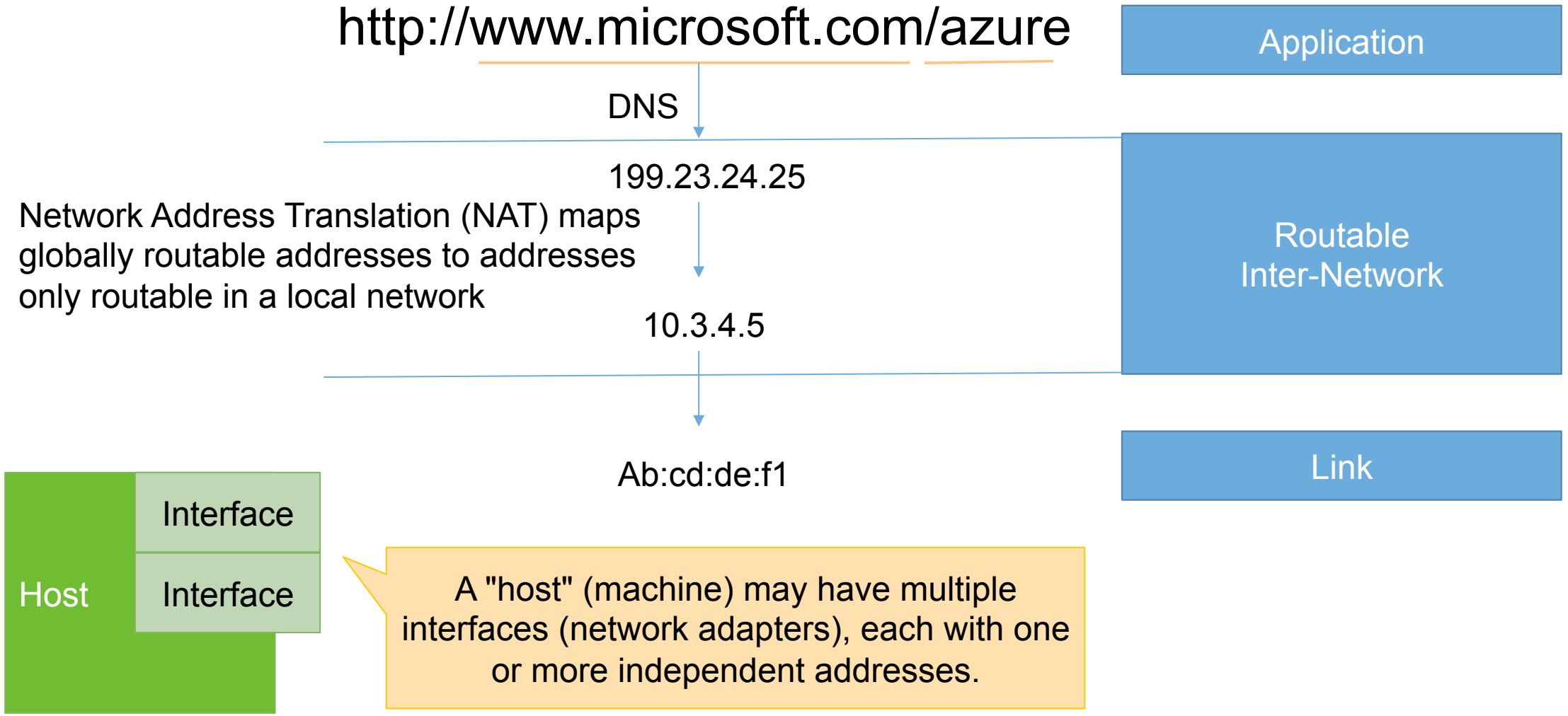
Interoperability

- Quality of the application and communication protocol
- Standardizing on
  - Addressing
    - Accomplished on communication protocol level
  - Framing
  - Message payload and semantics
    - Contract & schema
  - Delivery assurances
  - Security

Location Transparency

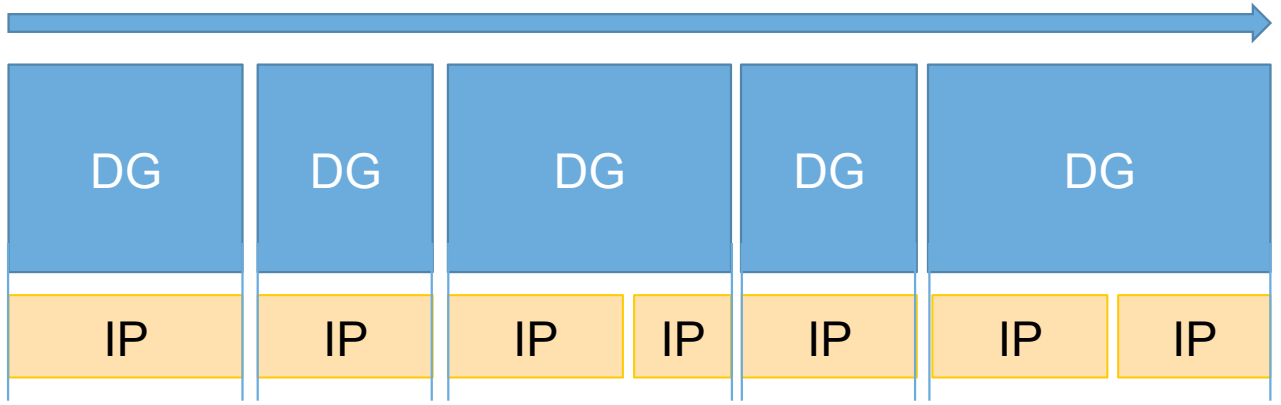
- Quality of communication protocol
- Standardizing the addressing
- Several degrees of transparency:
  - No transparency at all
    - Tight coupling
    - Service is expected to run at defined location that cannot change
  - Complete transparency
    - Service might be moved anywhere
    - Change some configuration data, if at all
  - Some transparency
    - Service might be moved within a cluster of servers or local network

Addressing

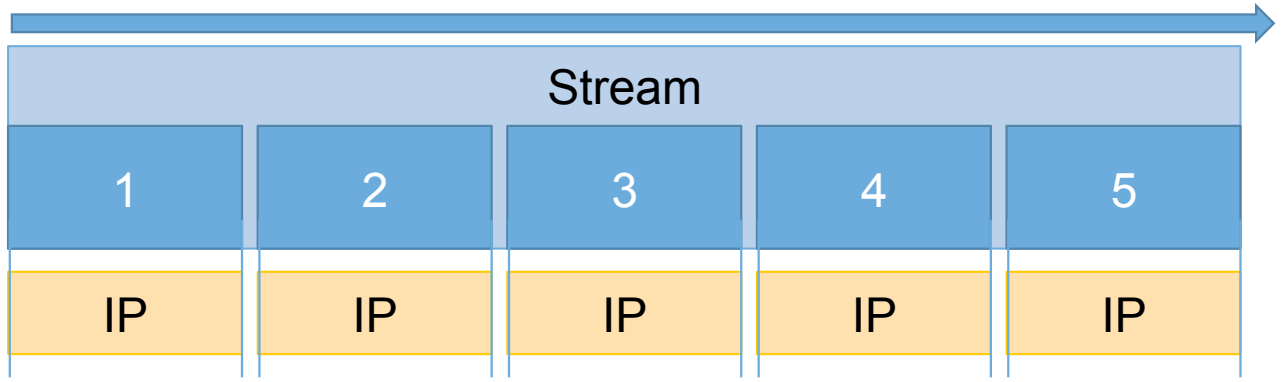

http://www.microsoft.com/azure
Application
DNS
199.23.24.25
Network Address Translation (NAT) maps globally routable addresses to addresses only routable in a local network
Routable Inter-Network
10.3.4.5
Ab:cd:de:f1
Link
Host
Interface
Interface
A "host" (machine) may have multiple interfaces (network adapters), each with one or more independent addresses.

Datagram and Stream Transports: UDP/TCP

- A "Datagram" is a data package not related to any other package
- UDP/IP is the (dominant) routable datagram protocol over IP
- "Best effort" transfer, no acknowledgement of delivery
- No congestion control; large packets can span IP frames, but whole packet is lost when one frame is lost



- A "stream" is an illusion of an unbounded and unstructured sequence of bits/bytes created over an ordered sequence of packets.
- TCP/IP is the (dominant) routable stream protocol over IP
- Acknowledged delivery, retransmission on packet loss, order enforcement
- Congestion control



Multi-Channelling

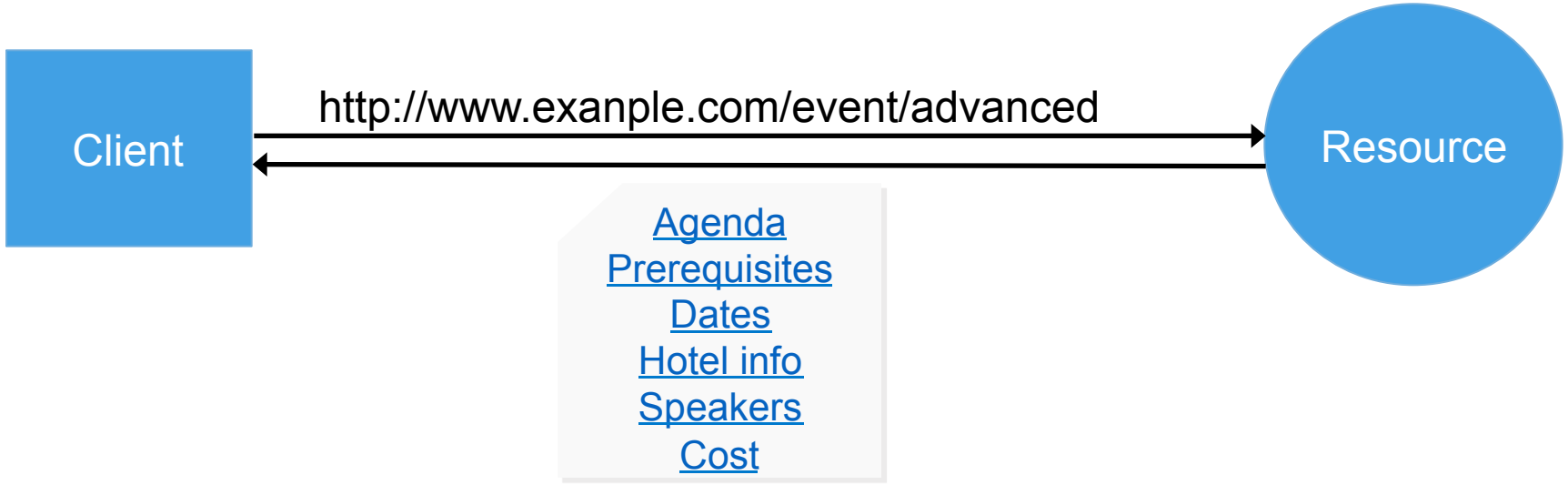
- A service can be reached through at least one channel
  - The host can provide the communication mechanisms
    - This is not strictly necessary!
- A service might listen to several channels
  - Having more than one edge
  - If contract is the same
- Service itself and its edges form different layers
- The edge is the services UI!

Loose Coupling

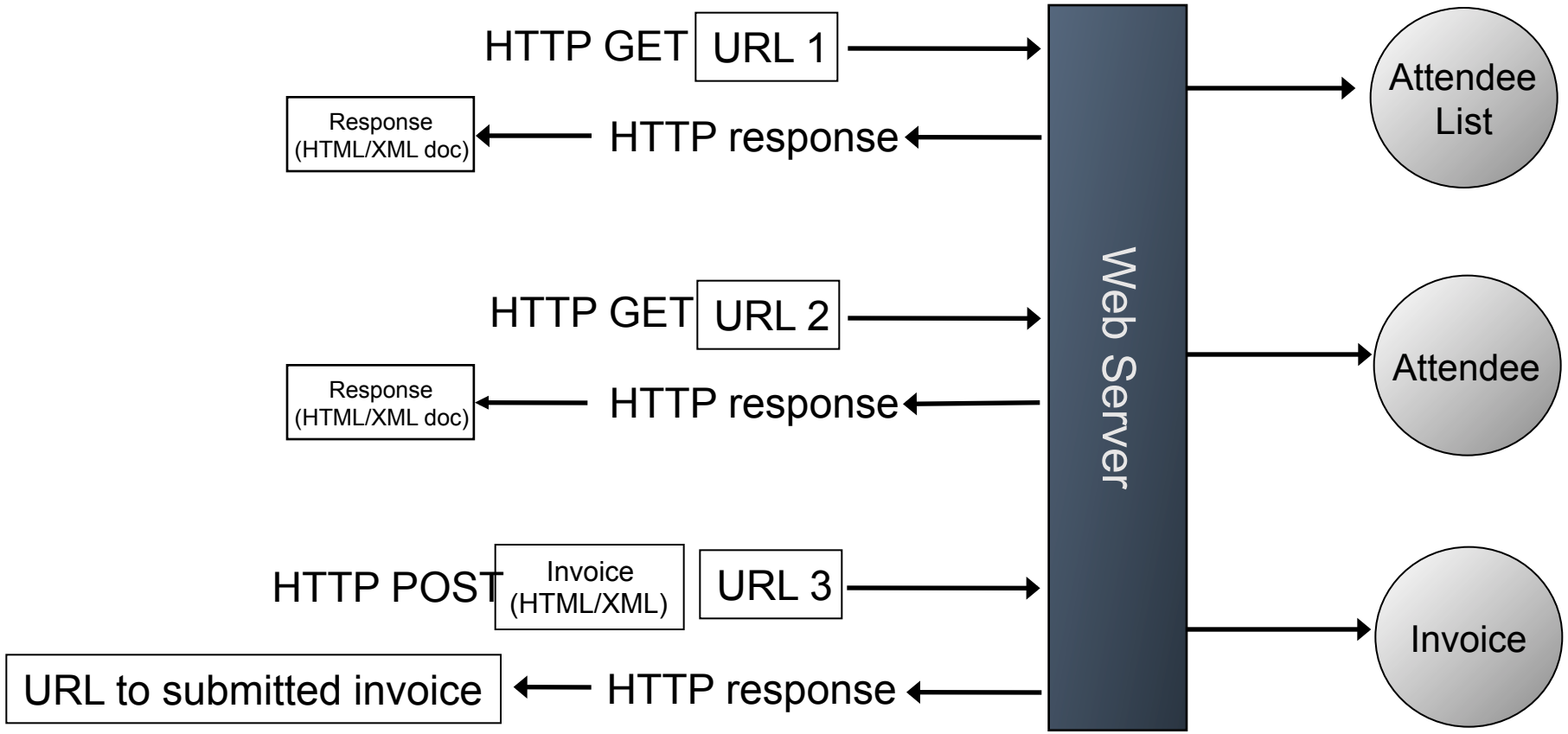
- Acquiring knowledge of a service while remaining independent of that service
- Accomplished through the use of service contracts
  - Fixed contract plus payloads (e.g. HTTP/REST + JSON)
  - Variable contracts (WSDL, etc.)
- Services interact within predefined parameters
- Advantages:
  - Supports reusability
  - Enables composability
  - Supports statelessness
  - Encourages autonomy

Representational State Transfer (REST)

- Client references a Web resource using a URL
- A representation of the resource is returned
- The representation puts the client into some state
- Dereferencing a hyperlink accesses another resource
- The new representation "transfers" the client application into yet another state



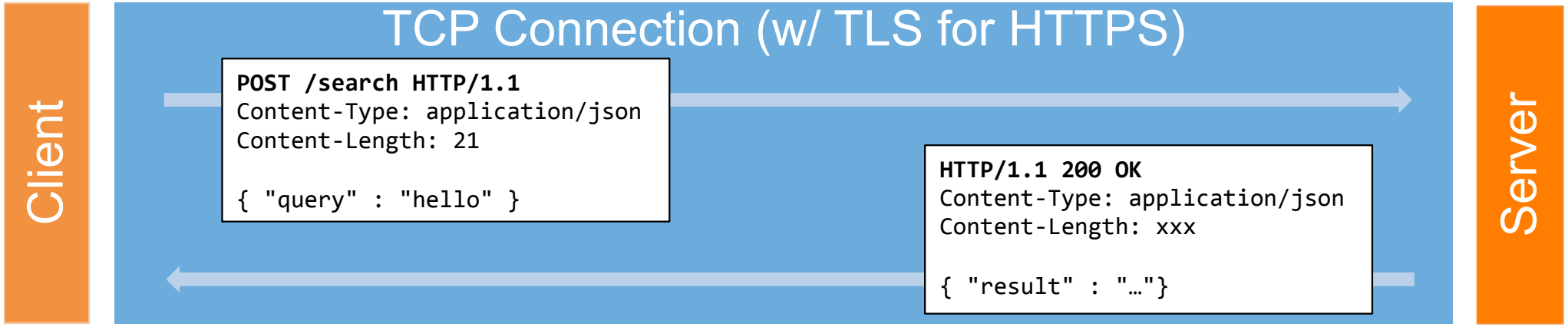
Conference Invoicing System



HTTP 1.1

Patterns	ReqResp
Symmetric	No
Multiplexing	No
Encodings	Variable
Metadata	Yes
Assurances	-

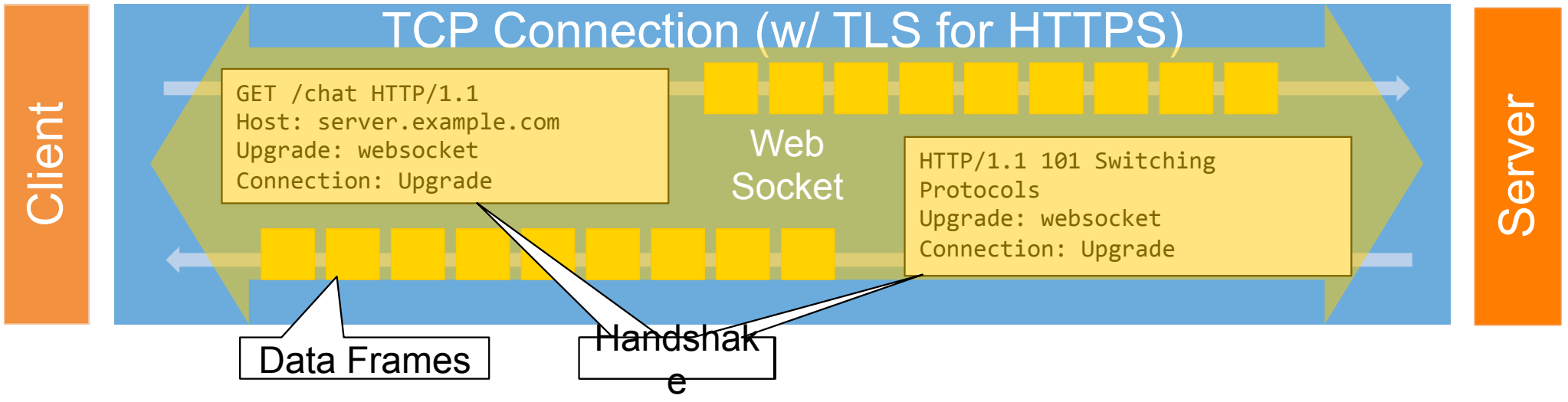
- HTTP 1.1 is the **Application Protocol** for the web
  - Simple structure, text based, ubiquitious
  - Client-initiated (asymmetric) request/response flow
  - No multiplexing
  - HTTP embodies the principles of "Representational State Transfer". REST is not a protocol, it's the architectural foundation for the WWW.



Web Sockets

Patterns	Duplex
Symmetric	No
Multiplexing	No
Encodings	Fixed
Metadata	No
Assurances	-

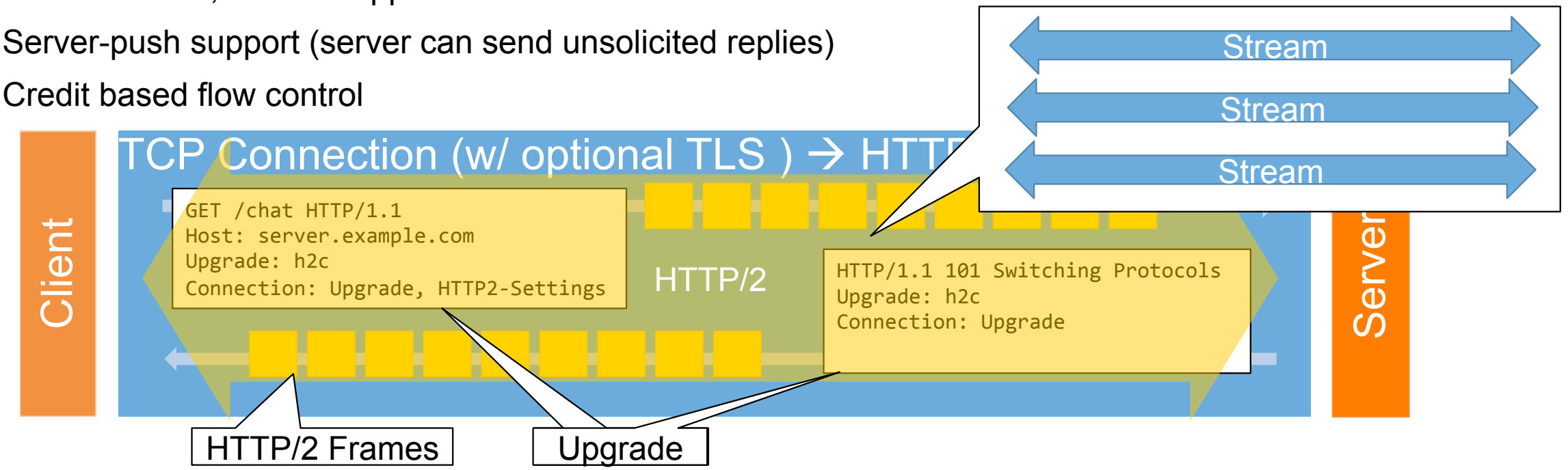
- Web Sockets is a **Stream Tunneling Protocol**
  - Allows using the HTTP 1.1 port (practically only HTTPS) for bi-directional, non-HTTP stream transfer
  - Web Sockets by itself is neither a Messaging or an Application Protocol, as it defines no encoding or semantics for the stream.
  - Web Sockets can tunnel AMQP, MQTT, CoAP/TCP, etc.



HTTP/2

Patterns	RR, OW/SC
Symmetric	No
Multiplexing	Yes
Encodings	Variable
Metadata	Yes
Assurances	-

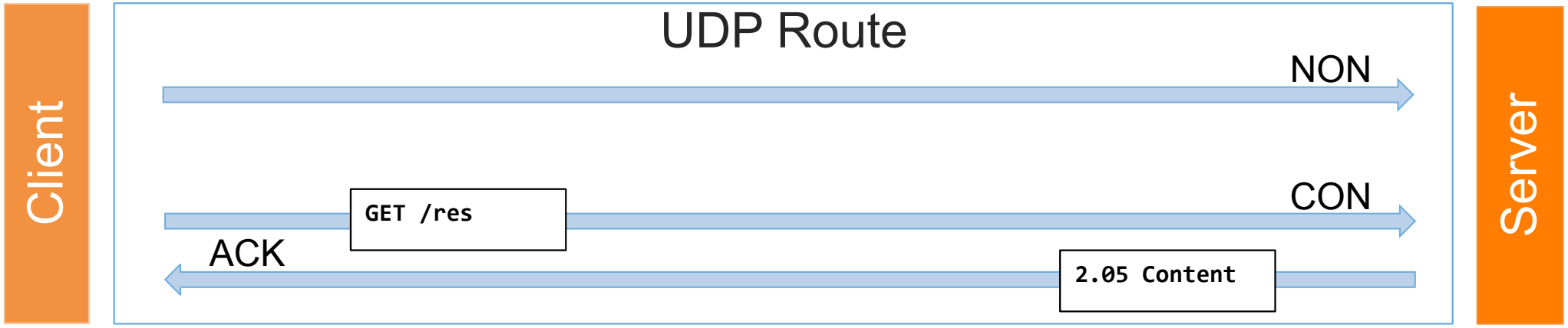
- HTTP/2 is an **Application Protocol**; successor of HTTP 1.1
  - Same semantics and message model, different implementation
  - Multiplexing support, binary standard headers, header compression.
  - Uses Web Socket like upgrade for backward compatible integration with HTTP 1.1, no WS support
  - Server-push support (server can send unsolicited replies)
  - Credit based flow control



CoAP
Constrained Application Protocol

Patterns	RR
Symmetric	Yes
Multiplexing	No
Encodings	Variable
Metadata	Yes
Assurances	-

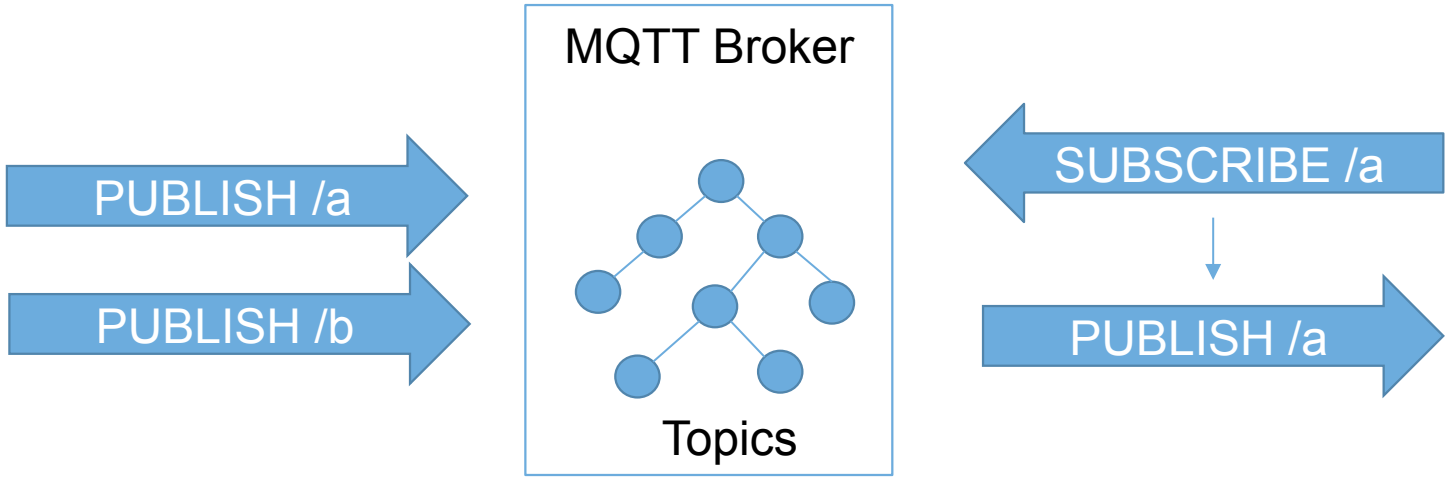
- CoAP is a lightweight **Application Protocol**
  - Adapts principles of HTTP to very constrained devices
  - CoAP is based on UDP, definition of CoAP for TCP is underway
  - Supports multicast on UDP
  - Creates a simple reliability layer over UDP using ACKs



MQTT

Patterns	Oneway
Symmetric	No
Multiplexing	No
Encodings	Fixed
Metadata	No
Assurances	AMO, ALO, EO

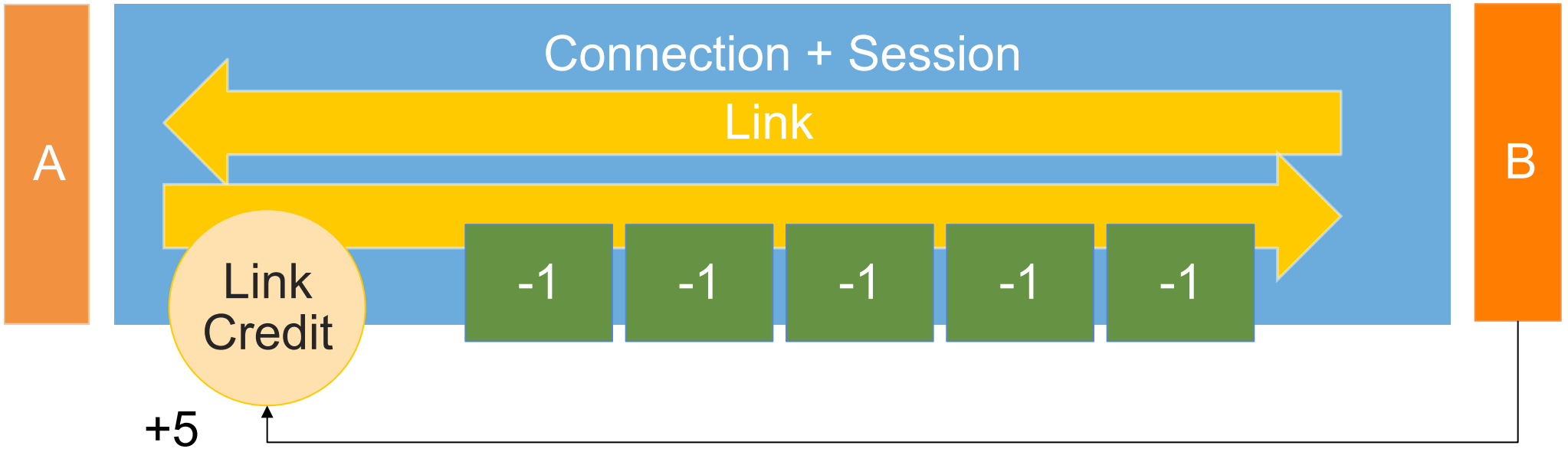
- MQTT is a lightweight **Publish and Subscribe Protocol**
  - Optimized for minimizing protocol overhead
  - Publish/Subscribe gestures in the protocol
  - One-way communication



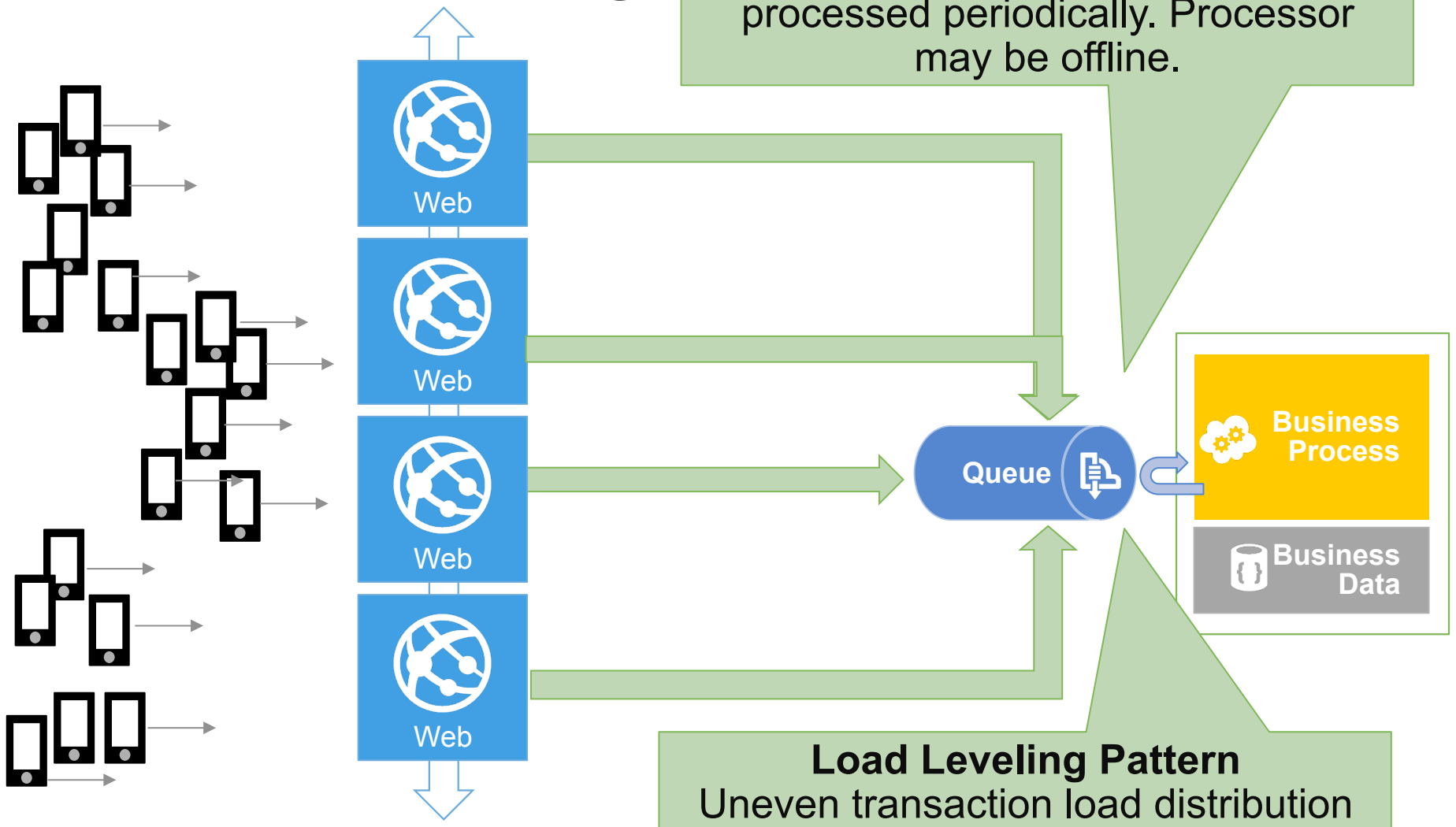
AMQP
Advanced Message Queuing Protocol

Patterns	Any
Symmetric	Yes
Multiplexing	Yes
Encodings	Variable
Metadata	Yes
Assurances	AMO, ALO, EO

- AMQP is a symmetric, reliable **Message Transfer Protocol** with support for multiplexing and flow control
  - Extensible, allowing for publish/subscribe and other gestures to be layered on top of the baseline protocol
  - Supports multiple security models



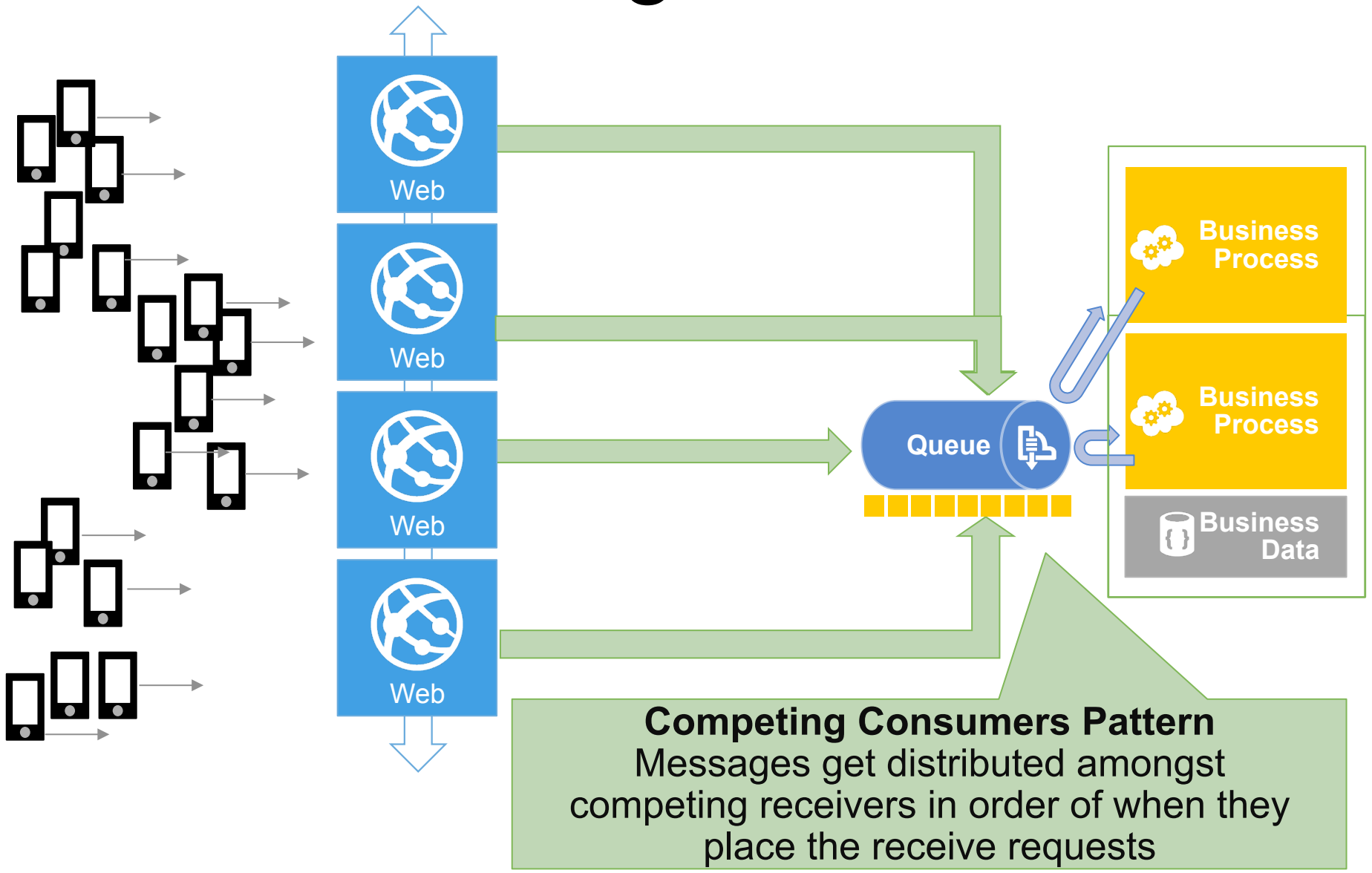
Load Leveling



Adding a message queue allows the business process to handle transactions at optimal capacity use and without getting overwhelmed

Spiky loads are buffered by the queue until the processor can handle them

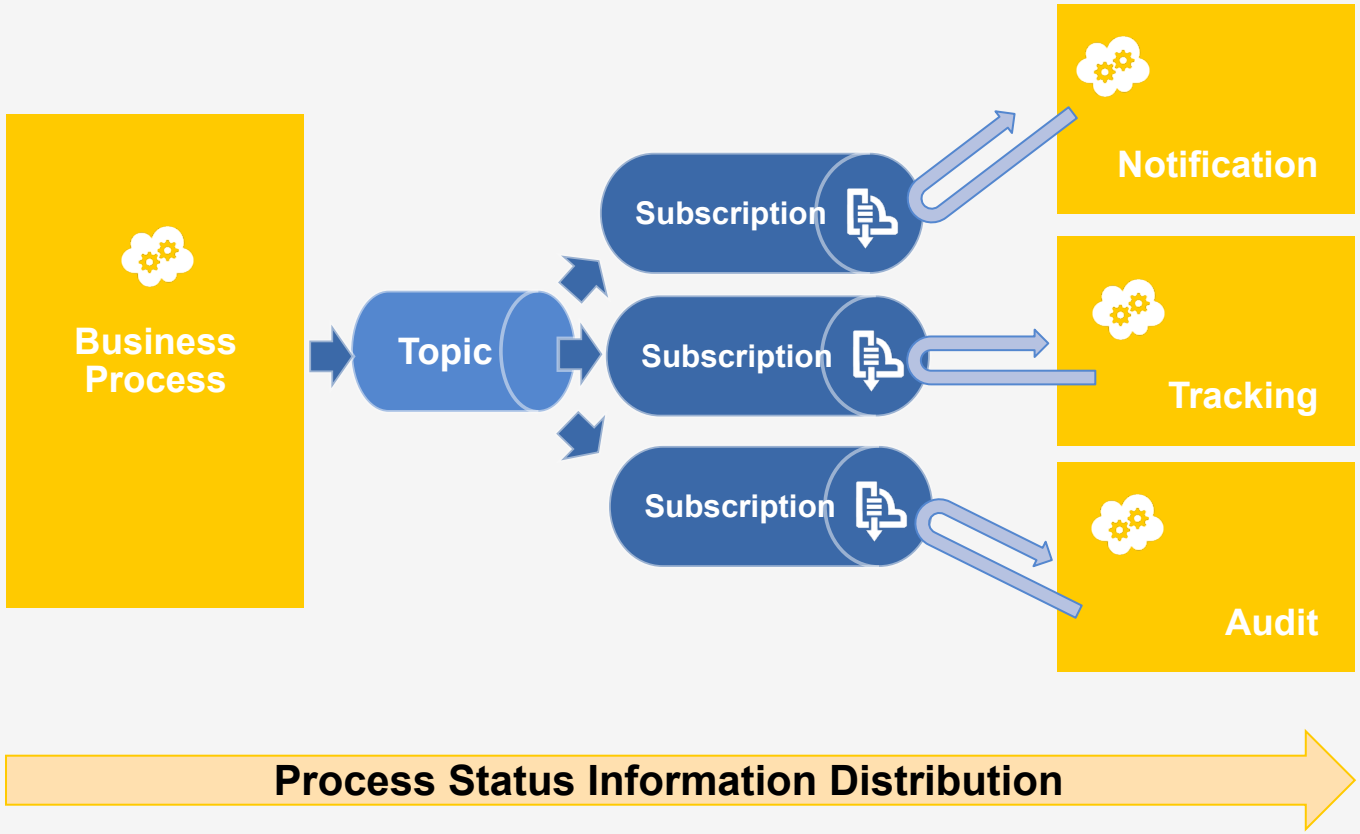
Load Balancing



Observing the queue length trends allows spinning up further resources as needed to handle exceptional load.

Publish/Subscribe with Topics

- Taps
  - Copies of main message flow for auditing or diagnostics purposes
- Multicast Fan-Out
  - Distribution of messages to multiple interested parties
- Filtering/Partitioning
  - Selective distribution of messages based on SQL'92 rules evaluated over message properties



Messaging Infrastructures

- “Enterprise” Message Brokers
  - Lots of features, transaction support, robust and durable publish/subscribe, multi-node clustering
  - Azure Service Bus, IBM MQ, Apache ActiveMQ, RabbitMQ
- Lightweight Message Brokers
  - Constrained features, compromising on capabilities for scale or on scale/robustness for compactness
  - Azure Queues, AWS SQS, Mosquitto
- Event Ingestors
  - Specialized brokers for telemetry capture
  - Azure Event Hubs, Apache Kafka, Kinesis

“Dumb Pipes” vs. ESBs

Thoughtworks TechRadar (2010!)

ESBs actively undermine the reasons for choosing the bus approach: low latency, loose coupling, and transparency.

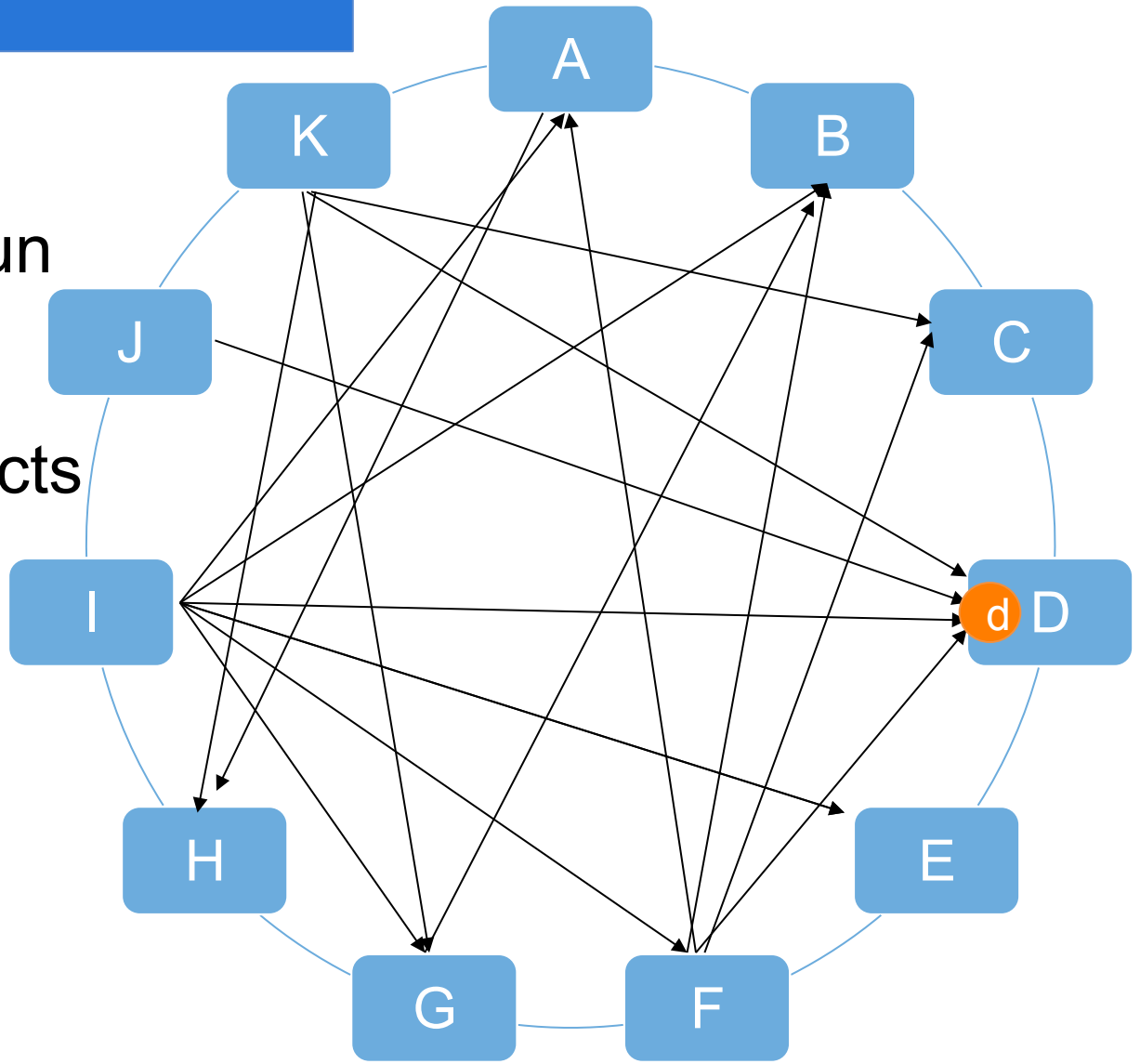
In contrast we have seen considerable success with Simple Message Buses where the integration problems are solved at the end points, rather than inside a vendor ESB system.

IT departments are increasingly striving to liberate data from disparate systems. A broad set of approaches have been promoted under the generic term Service Oriented Architecture (SOA). This has led to confusion about what the term and approach actually means. We believe businesses do not need the complex enterprise service bus products advocated by vendors. ESBs actively undermine the reasons for choosing the bus approach: low latency, loose coupling, and transparency.
In contrast we have seen considerable success with Simple Message Buses where the integration problems are solved at the end points, rather than inside a vendor ESB system. The most well known Simple Message Bus approach is one based on the principles of REST and leveraging the proven scalability of the web. However organizations that have already invested in ESB infrastructure can leverage the useful parts of that infrastructure (reliable messaging etc) while still using a Simple Message Bus approach and performing integrations at the edges of the system.

Autonomy and Integration

- Autonomous Services
- Independently Deployed and Run
- Independent Teams
- Shipping Features/Fixes Daily
- Services Own and Fulfill Contracts

- Messaging Infrastructure is lines, not boxes
- Integration occurs at the edges, not in the center



Summary: Edge Principles

- Business logic and edge are separate layers and potentially tiers
- Explicit boundary
- Conceptually one single interface
  - Read/write pairs
  - Different channels, potentially paralle channels
- Asynchronous calls
- Message driven
- Location transparency
- Loosely coupled
- Separate hosts

Summary: Generalized Architecture Model