Software Architecture

DIT344

Sam Jobara Ph.D.
jobara@chalmers.se
Software Engineering Division
Chalmers | GU

Architectural Styles
Part III

Learning Objectives

- Introduce three current distributed architecture styles
- Look at the relationships between style and attributes.
- Provide implementation tactics and cases for these styles
- Discuss these styles key selection criteria

Main Reference:

Fundamentals of Software Architecture, an engineering approach

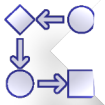
by Neal Ford; Mark Richards *Published by O'Reilly Media, Inc., 2020*

Agenda

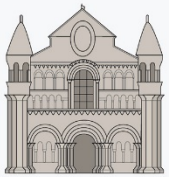

Warm up in Style

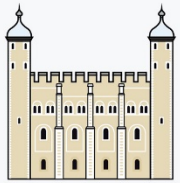

Peer-to-Peer Style


Microservices Style


Event-Driven Style


Choosing a Style


ROMANESQUE


NORMAN

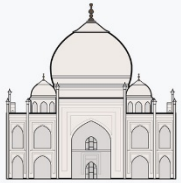

GOTHIC


MEDIEVAL


RENAISSANCE


TUDOR


ELIZABETHAN


INDOISLAMIC


BAROQUE


JACOBEAN

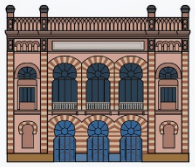

PALLADIAN

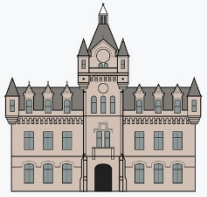

ROCOCO


GEORGIAN


NEOCLASSICAL

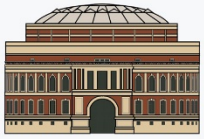

GOTHIC REVIVAL

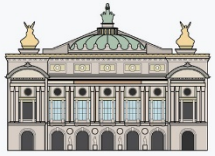

MOORISH REVIVAL

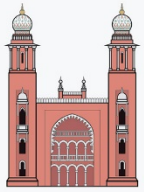

BARONIAL


FEDERAL


REGENCY


ITALIANATE


EMPIRE


INDOSARACENIC


JACOBETHAN


CHICAGO SCHOOL

Architectural style defined

Architectural patterns/styles and tactics are ways of capturing proven good design structures, so that they can be reused for similar projects context.

An Architecture style is a set of rules and constraints that prescribe

- vocabulary/metaphor:

  which types of components, interfaces & connectors must/may be used in a system. Possibly introducing domain-specific types

- structure:

  How components and connectors may be combined

- behavior:

  How the system behaves

- guidelines:

  These support the application of the style
  (how to achieve certain system properties)

Architectural style defined

- Architectural styles are design paradigms for a set of design dimensions

  Some architectural styles emphasize different aspects such as: Subdivision of functionality, Topology or Interaction style

- Styles are open-ended; new styles will emerge
- A single architecture can use several architectural styles
- Architectural styles are not disjoint, they can exist in hybrid mix
- Reusability and use cases define popularity of certain styles

Architectural style types

Monolithic vs. Distributed Architectures

Architecture styles are two types: *monolithic* (single deployment unit of all code) and *distributed* (multiple deployment units connected through access protocols).

Monolithic

- Layered architecture (*n*-tier or client-server architecture)
- Pipeline architecture
- Microkernel architecture

Distributed

- Peer-to-Peer architecture
- Microservices architecture
- Event-driven architecture
- Service-oriented architecture

Architectural style types

Distributed architectures all share a common set of challenges and issues not found in the monolithic architecture styles *Monolithic*

1- The network is not reliable

2- Latency is not zero (microsecond vs. millisecond)

3- Bandwidth is not infinite

4- The network is not secure

5- The network topology always changes (unpredicted performance)

6- There are many network administrators, not just one

7- The network is not homogeneous or a pure style

TECHNIQUE FOR ARCHITECTURE DESIGN

This technique consists of five steps that are performed iteratively:

1. Identify architecture objectives.

Provide scope of design

2. Identify key scenarios.

Key scenarios represent architecture drivers, significant use cases, intersections between quality attributes and functionality, or tradeoffs between quality attributes.

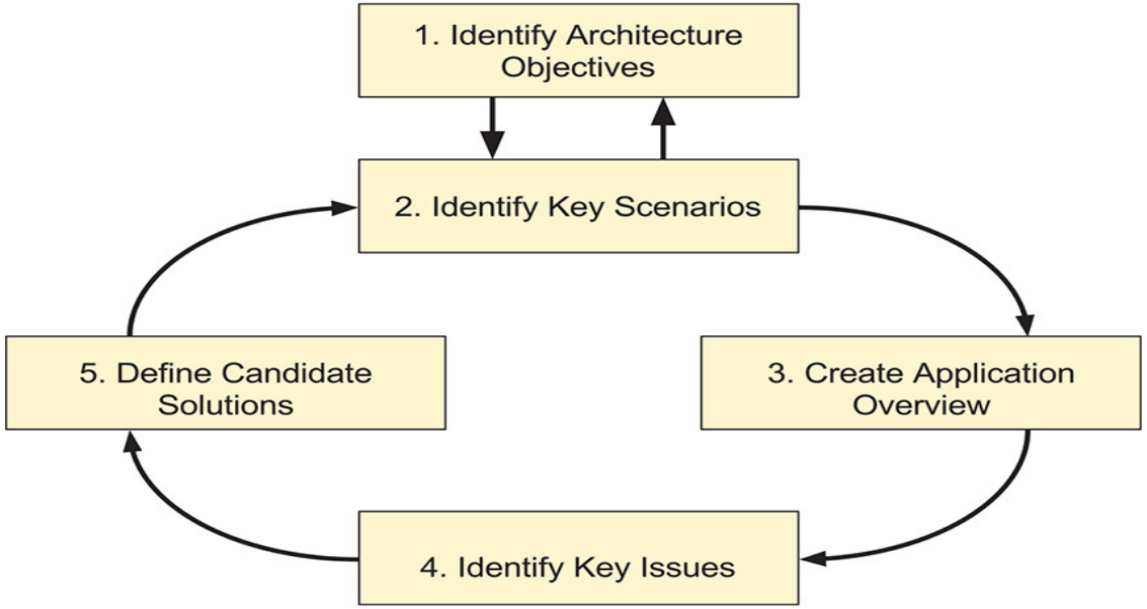


FIGURE 7.5 Iterative steps of the technique for architecture and design

3. Create application overview.

This step is divided into the following set of activities:

a. Determining application type, **b.** Identifying deployment constraints, **c.** Identifying important architecture design styles, **d.** Determining relevant technologies

4. Identify key issues.

Such as quality attributes and crosscutting concerns. Crosscutting concerns are features of the design that may apply across all layers, components, and tiers, such as the following:

a. Authentication and authorization, **b.** Caching, **c.** Communication,
d. Configuration management, **e.** Exception management, **f.** Validation and testing

5. Define candidate solutions.

Candidate architectures include an application type, deployment architecture, architectural style, technology choices, quality attributes, and crosscutting concerns.

Synch. vs. Asynch. & Decoupling

Synchronous calls between two distributed services have the caller wait for the response from the callee (real-time chat) .

Asynchronous calls allow fire-and-forget (or choose when to respond like sensors) semantics in event-driven architectures, allowing two different services to differ in operational architecture

What does *Decoupled Architecture* mean?

It is a type of computing architecture that enables computing components or layers to execute independently while still interfacing with each other.

Decoupled architecture is also used in software development to develop, execute, test and debug application modules independently. Cloud computing architecture implements decoupled architecture where the vendor and consumer independently operate and manage their resources.

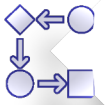
Decoupled architecture helps achieve higher computing performance, deployment, reusability, and testability by isolating and executing individual components independently and in parallel.

Agenda

Warm up in Style

Peer-to-Peer Style

Microservices Style

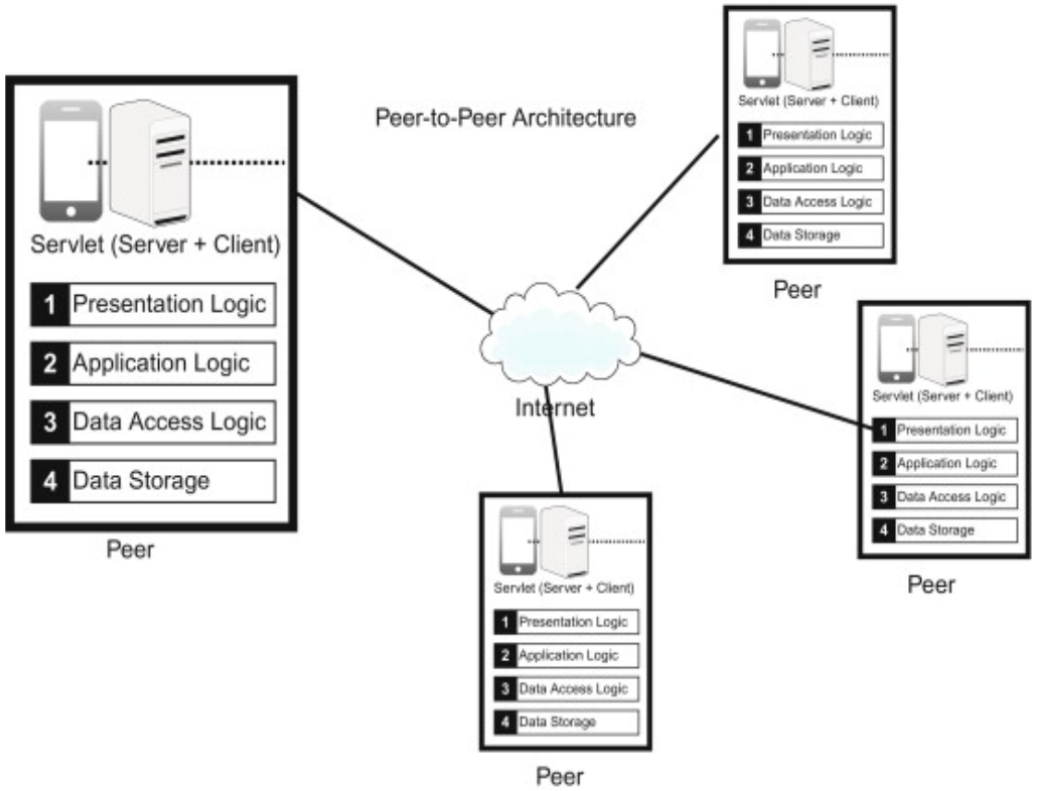


Event-Driven Style

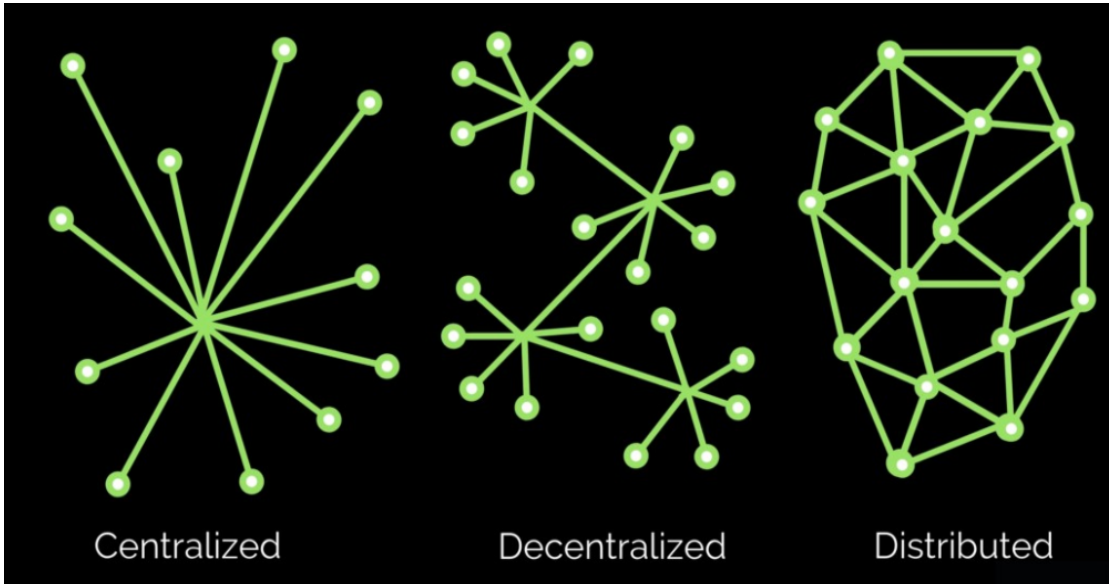
Choosing a Style

P2P Architecture Style

- Peer-to-peer (P2P) is a distributed computing architecture
- Divides tasks or workloads across several computer systems (of nodes or peer).
- P2P networks can be used to share any kind of digital assets, such as data, or smart contracts.
- The structure of a pure P2P network is sustained by its users, who can provide governance and use resources.
- A single peer can be an independent client-server Servlet structure.



How P2P is different?



💪 Fault tolerance:
- Low: Centralized systems
- Moderate: Decentralized systems
- High: Distributed systems

🔧 Maintenance:
- Low: Centralized systems
- Moderate: Decentralized systems
- High: Distributed systems

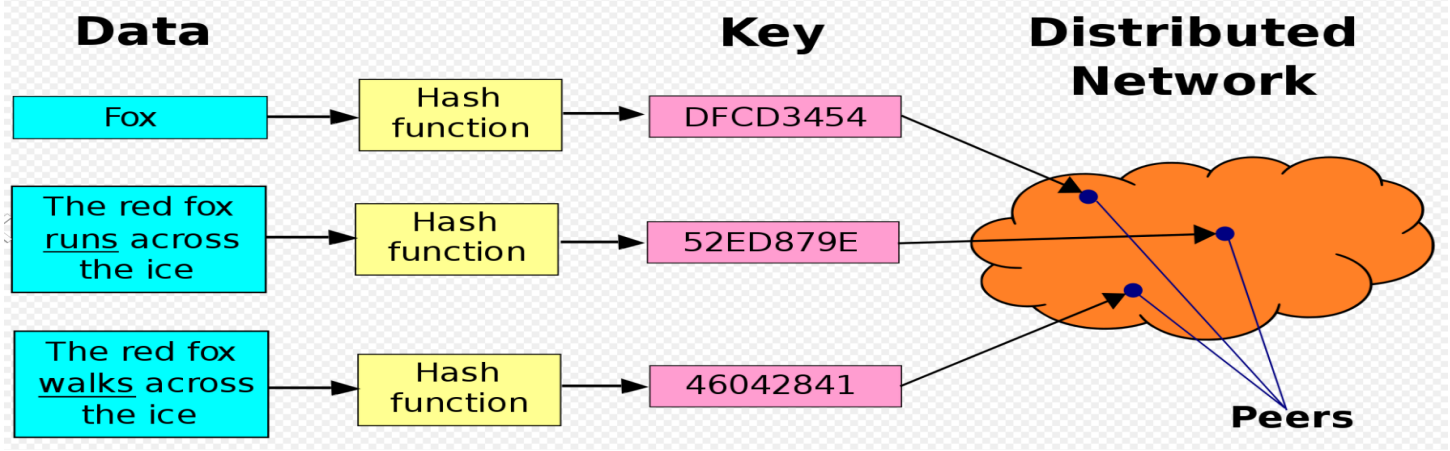
🚀 Scalability:
- Low: Centralized systems
- Moderate: Decentralized systems
- High: Distributed systems

Decentralized Systems:
Every node makes its own decision. The final behavior of the system is the aggregate of the decisions of the individual nodes. Note that there is no single entity that receives and responds to the request. eg. blockchain,

Distributed Systems:
Means that the processing is shared across multiple nodes, but the decisions may still be centralized and use complete system knowledge. eg. AWS, Cloud Instances, Google, Facebook, Netflix, etc.

Structured peer-to-peer networks

- In *structured peer-to-peer networks* the overlay is organized into a specific topology, and the protocol ensures that any node can efficiently search the network for a file/resource.
- The most common type of structured P2P networks implement a distributed hash table (DHT), in which a variant of consistent hashing is used to assign ownership of each file to a particular peer. This enables peers to search for resources on the network using a hash table: that is, (*key*, *value*) pairs are stored in the DHT.
- DHT distribute responsibility of storing & retrieve data in large **network.**



P2P Implementations

- Windows 10 updates are delivered both from Microsoft's servers and through P2P (bandwidth sharing)
- The decentralized framework of P2P systems makes them highly available and resistant to cyber attacks and also more scalable.
- The more users join it, the more resilient and scalable it gets. Bigger P2P networks achieve high levels of security because there is no single point of failure.
- The peer-to-peer architecture popular examples with varying use cases include:
  - BitTorrent (file-sharing)
  - Tor* (anonymous communication software),
  - Many more decentralized apps (See Blockchain lecture)

*The Onion routing is implemented by encryption in the application layer of a communication protocol stack, nested like the layers of an onion. Tor encrypts the data, including the next node destination IP address, multiple times and sends it through a virtual circuit comprising successive, random-selection Tor relays.

P2P Implementations

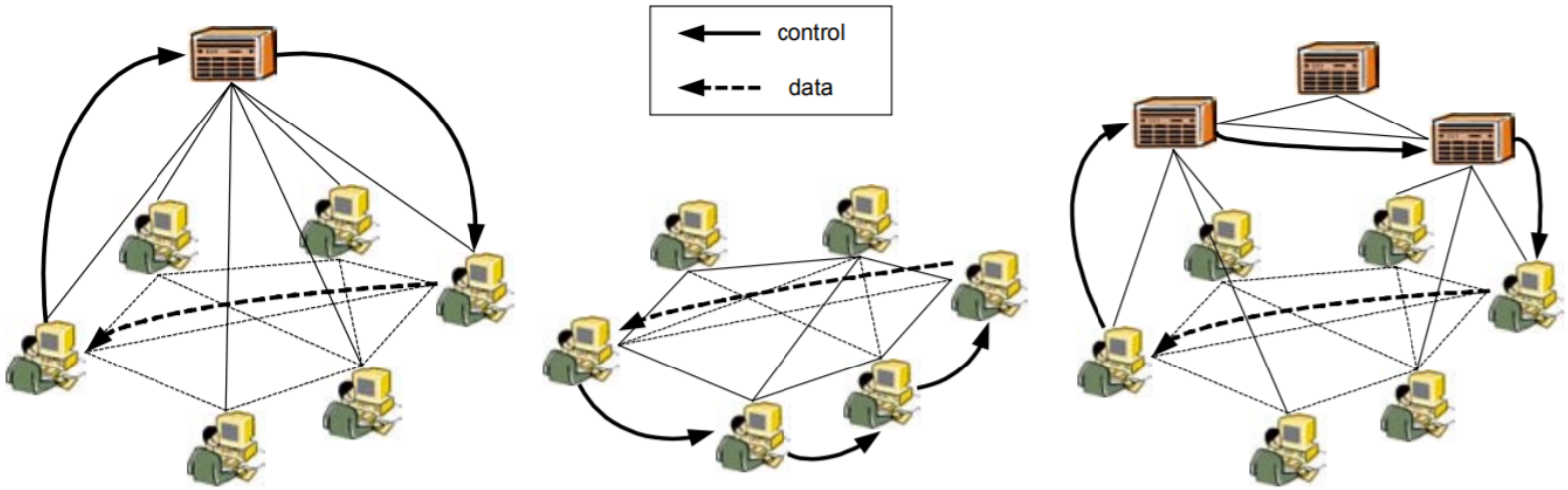
- Unstructured vs. Structured (public vs. private) MPLS Multi-protocol label switching
- Security, Resilience and Scalability concerns
- Controlling authority (pure vs. hybrid)

dApps can run on both a P2P network as well as a blockchain network.

It is challenging to achieve pure dApps network due to the need of some governance (to be covered in blockchain)

P2P Architectural styles

We present here three popular P2P styles*



Three peer-to-peer architectures: mediated (left), pure (middle) and hybrid (right)

***A comparison of peer-to-peer architectures**

Peter Backx, Tim Wauters, Bart Dhoedt, Piet Demeester

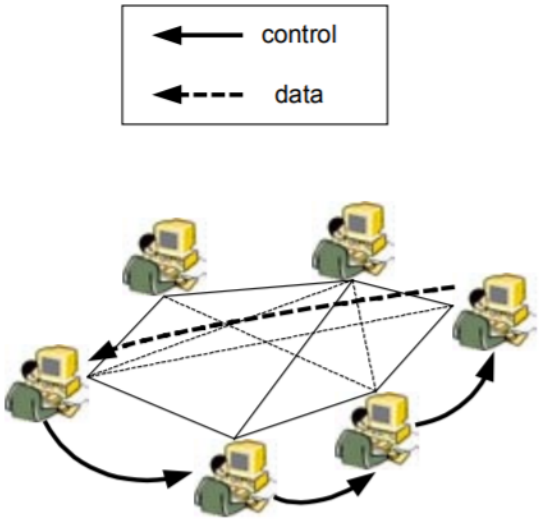
Broadband Communication Networks Group (IBCN),

Department of Information Technology (INTEC), Ghent University, Belgium

P2P Architectural styles

Pure peer-to-peer architecture

- Applications will not use a central server at all (except possibly for logging onto the network).
- Queries for files can be flooded through the network or more intelligent mechanisms can be used.
- Have become quite unpopular because they generate a lot of overhead traffic to keep the network up and running.
- Some adopters still use this model because it offers an unprecedented anonymity, not found in any other architecture.



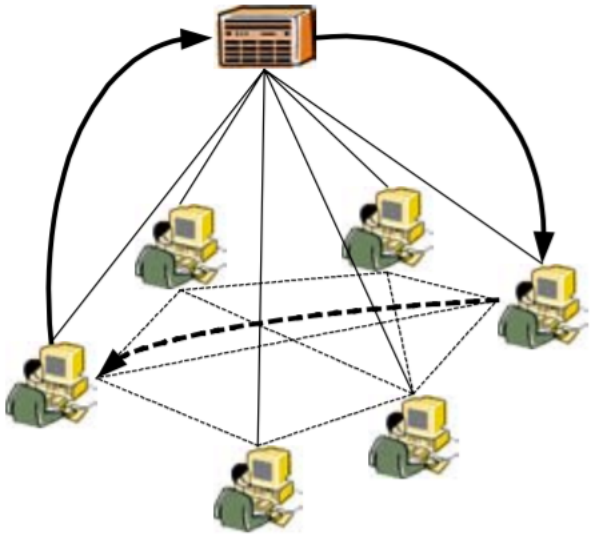
P2P Architectural styles

Mediated architecture

Uses a client-server setup for its control operations. All peers log on to a central server that manages the file and user databases.

Searches for a file are sent to the server and, if found, the file can be downloaded directly from a peer.

In most cases the server will have a database of files shared by peers. Afterwards the server functions as a proxy that distributes the searches towards the peers.



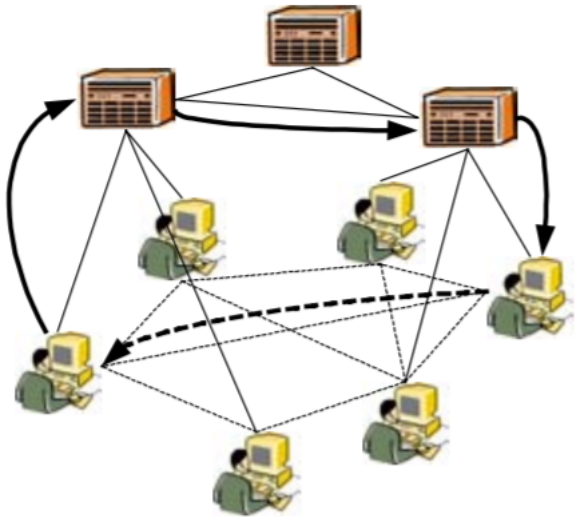
P2P Architectural styles

Hybrid architectures

Hybrid architectures introduce two layers in the control plane:

one of "normal" peers connecting to ultrapeers in a client-server fashion and one of ultrapeers connected with each other via a pure peer-to-peer network.

Both pure and hybrid architectures build an overlay network over the existing IP network.



P2P Architectural styles

The table below gives a broad overview of some distinguishing features of several peer-to-peer file sharing applications.

	Type of files	Community	Resume	Multisource downloading / swarming	Data integrity	Ano-nymity
Audio-Galaxy [11]	MP3	Strong (User groups, browsing, etc.)	Yes	No	Hash	No
Gnutella [12]	Everything	Weak (browsing)	Yes (on some clients not automatic)	Some clients (Xolox)	Work in progress	No
FastTrack [13]	Everything	Weak (browsing & messaging)	Yes	Yes	Hash	No
WinMX [14]	Everything	Weak (browsing & messaging)	Yes	Yes	Hash	No
SoulSeek [15]	Everything	Strong (individual and group chat, browsing)	Yes	No	None	No
FreeNet [5]	Everything	Application specific			Hash	Yes
eDonkey [16]	Everything	Weak	Yes	Yes	Hash	No

BitTorrent (file-sharing)
Tor (anonymous communication software),

P2P Architectural styles

P2P implementation considerations

P2P style Advantages	P2P style Disadvantages
Resilient, Highly available	No central governance
Cost effective, less overhead	Risk of data integrity
Less complex, easy to deploy	Security exposure
Allow for bandwidth sharing	Sensitive to network performance
Flexible and faster enquiries	Less practical without central log
Flexible hybrid models	May force you to upload files
Enable anonymity	May include illegal content
Support Decoupling	Not suitable for small systems

P2P Architectural styles

Learning Unit Obligation

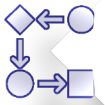
- Understand the Architecture style of P2P
- Realise the different topologies and use of P2P
- Understand the strength and weaknesses of P2P
- Realise how to mitigate P2P shortcomings

Agenda

Warm up in Style

Peer-to-Peer Style

Microservices Style



Event-Driven Style

Choosing a Style

Microservices Style

Extremely popular architecture style that has gained significant momentum in recent years due to mobile and cloud computing.

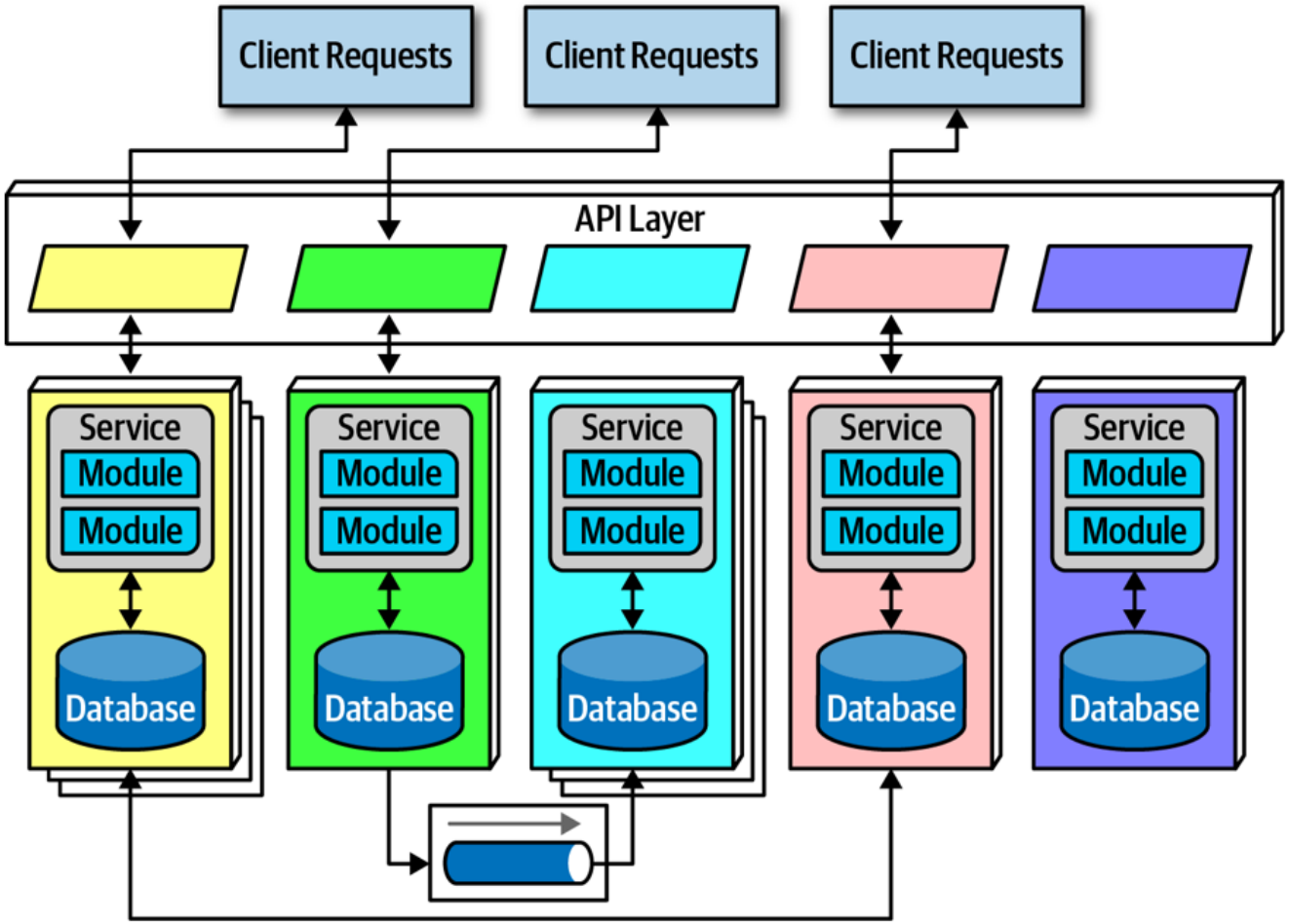


Figure 17-1. The topology of the microservices architecture style

According to a recent O'Reilly radar survey on the growth of cloud computing, one of the more interesting metrics stated that 52 percent of the 1,283 responses say they use microservices concepts, tools, or methods for software development.

Microservices Style-Features

- Microservices form a *distributed architecture*: each service runs in its own process, which originally implied a physical computer but quickly evolved to virtual machines and containers.

- Decoupling the services to this degree allows for a simple solution to a common problem in architectures that heavily feature multitenant infrastructure for hosting applications. Now, however, with cloud resources and container technology, teams can reap the benefits of extreme decoupling, both at the domain and operational level.

- Granularity correct granularity for services in microservices, and often make the mistake of making their services too small, which requires them to build communication links back between the services to do useful work. *The term "microservice" is a label, not a description.* Designing the right level of service component granularity is one of the biggest challenges within a microservices architecture.

Microservices Style-Features

Performance is negative side effect of the distributed microservices. Network calls take much longer than method calls, and security verification at every endpoint adds additional processing time, requiring architects to think carefully about the implications of granularity when designing the system.

Microservices is a distributed architecture. Architects advise against the use of transactions across service boundaries, making determining the granularity of services the key to success in this architecture.

Bounded Context. The driving philosophy of microservices is the notion of *bounded context*: each service models a domain or workflow. Thus, each service includes everything necessary to operate within the application, including classes, other subcomponents, and database schemas.

Microservices adopt a domain-partitioned architecture to the extreme.
Each service is meant to represent a domain or subdomain; it is domain-driven design.

Microservices Style-Features

Data Isolation. Another requirement of microservices, driven by the bounded context concept, is data isolation. Many other architecture styles use a single database for persistence. However, microservices tries to avoid all kinds of coupling, including shared schemas and databases used as integration points.

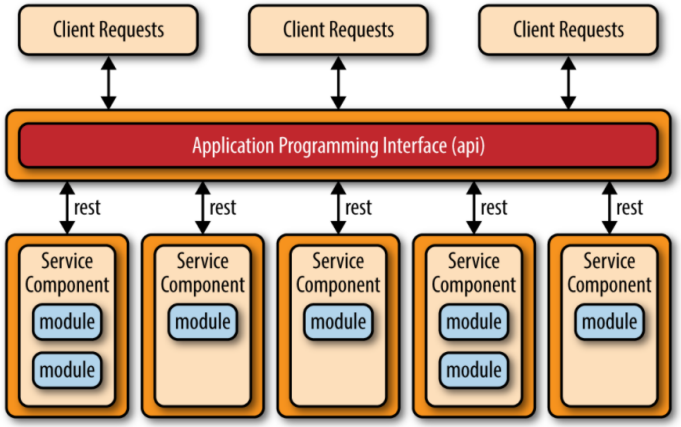
API Layer API layer is sitting between the consumers of the system. The API gateway will handle a large amount of the communication and administrative roles, allowing the microservices to remain lightweight. They can also authenticate, cache and manage requests, as well as monitor messaging and perform load balancing as necessary.

CD of DevOps The evolutionary path from monolithic applications to a microservices architecture style was prompted primarily through the development of continuous delivery, the notion of a continuous deployment pipeline from development to production which streamlines the deployment of applications.

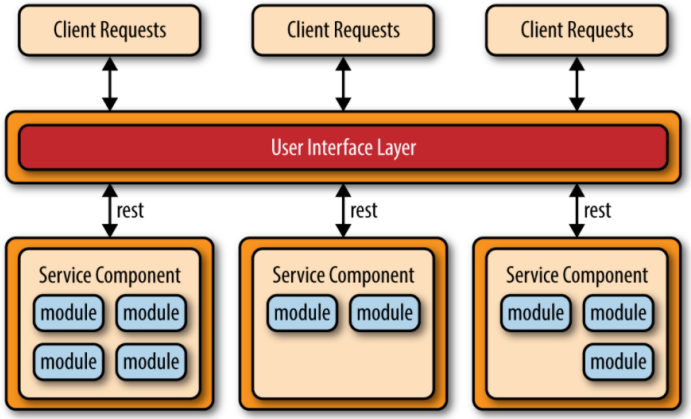
see A2

Microservices Style-Interface

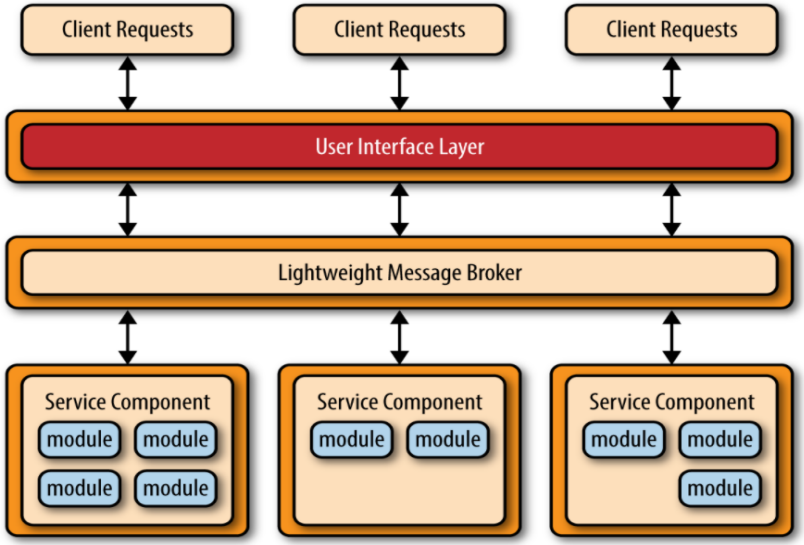
A key concept within the microservices architecture style is that it is a distributed architecture, all the components within the architecture are fully decoupled from one other and accessed through some sort of remote access protocol. The distributed nature of this style is behind its superior scalability and deployment characteristics.



Traditional web-based or fat-client business application



Web application remotely accesses separately deployed service.

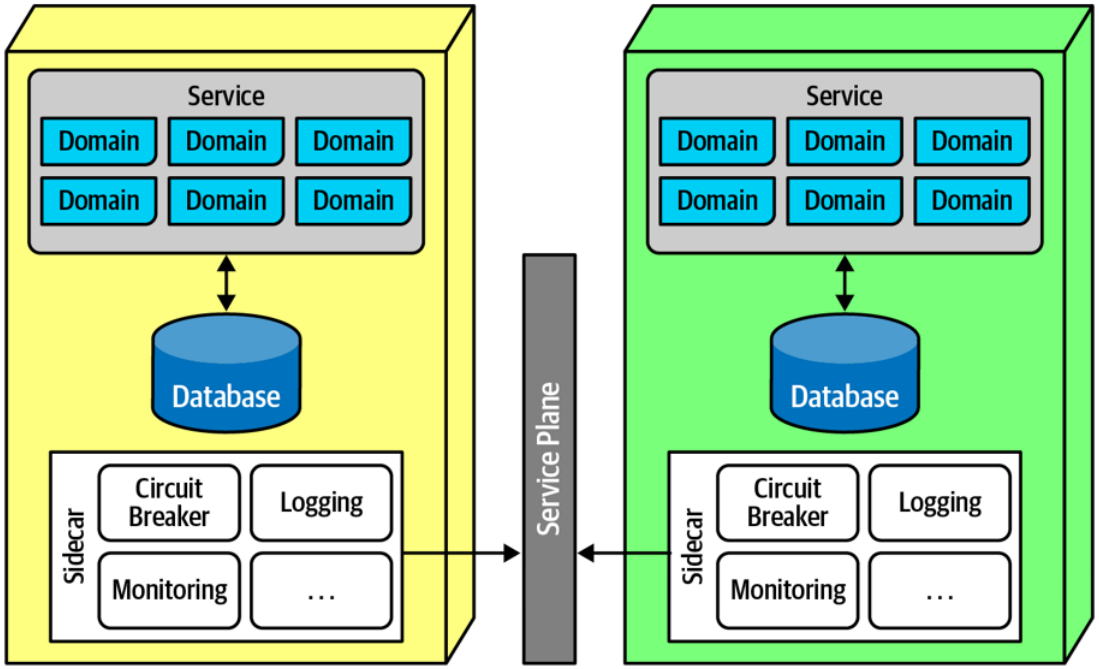


lightweight centralized message broker to access remote service components

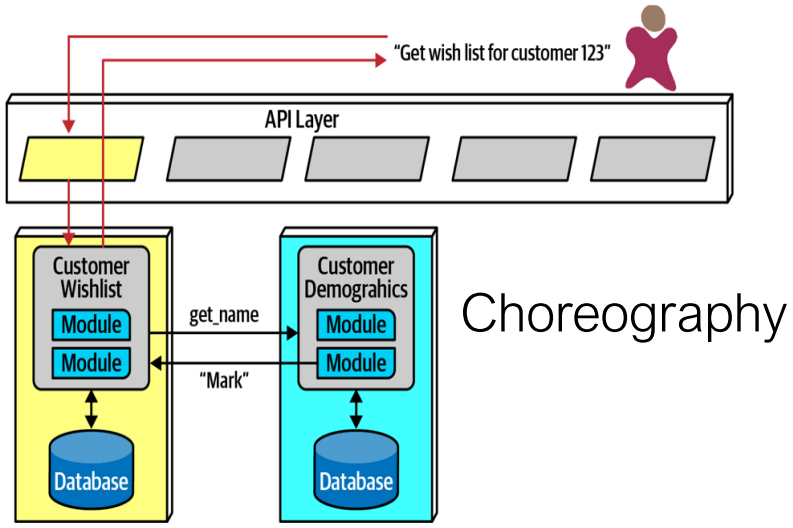
Rest: Remote access protocol

Microservices Style-Sidecar

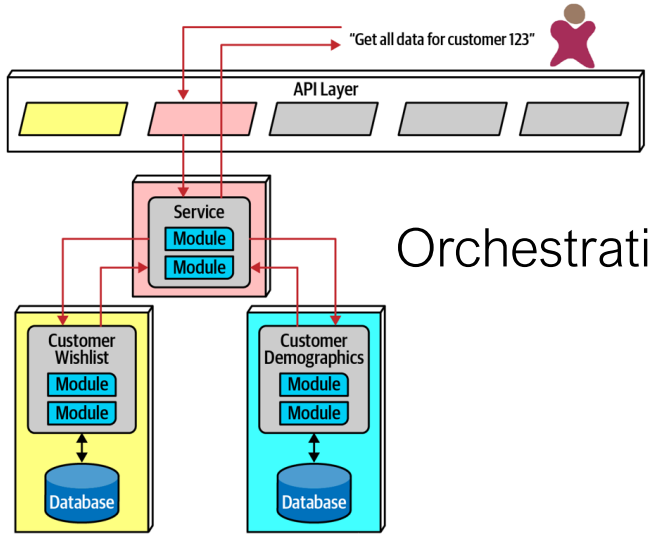
When a common operational concerns appear within each service as a separate component, the sidecar component can handle all the operational concerns that teams benefit from coupling together. Thus, when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, and each microservices receives that new functionality. The common sidecar components connect to form a consistent operational interface across all microservices



Microservices Style-Communication



Choreography



Orchestration

Choreography: utilizes the same communication style as a broker event-driven architecture. In other words, no central coordinator exists in this architecture, respecting the bounded context philosophy. Thus, architects find it natural to implement decoupled events between services.

Orchestration the developers create a service whose sole responsibility is coordinating the call to get all information for a particular customer. The user calls the ReportCustomerInformation mediator (light weight message broker).

Microservices Style

Architecture Characteristics

Communications between services in a microservices architecture can be:

- decentralized and synchronous
- choreographed and asynchronous
- orchestrated and synchronous/asynchronous.

In a decentralized and synchronous communications pattern, each service receives flow control, makes subsequent synchronous calls to other services and passes control to the next service.

In choreographed and asynchronous service communications, the service publishes events to a central message queue that distributes those events.

The centralized orchestration, enables both synchronous and asynchronous communication. The orchestrator sequences the various service calls based on a defined workflow.

Microservices Style

Architecture Characteristics

Offers high support for modern engineering practices such as automated deployment, and testability.

Microservices couldn't exist without the DevOps revolution and the relentless march toward automating operational concerns.

Fault tolerance and reliability are impacted when too much interservice communication is used. independent, single-purpose services generally lead to high fault tolerance.

High scalability, elasticity, and evolutionary systems utilized this style to great success.

The architecture relies heavily on automation and intelligent integration with operations, developers can also build elasticity support into the architecture.

Microservices Style

Architecture Characteristics

Favors high decoupling at an incremental level, it supports evolutionary change.

By building an architecture that has extremely small deployment units that are highly decoupled, that can support a faster rate of change. Make many network calls to complete work, which has high performance overhead, and also invoke security checks to verify identity and access for each endpoint.

Many patterns/tactics exist in the microservices world to increase performance, including intelligent data caching and replication to prevent an excess of network calls.

Performance is another reason that microservices often use choreography rather than orchestration, as less coupling allows for faster communication and fewer bottlenecks.

Microservices Style

Microservices style implementation considerations

MS style Advantages	MS style Disadvantages
High scalability, & agility	Overall high cost
High reliability	Performance bottlenecks
High deployability/testability	Can get complex
High Fault tolerance	High overhead for security
Very high modularity	Hybrid tactics needed
Automation & Integration	

P2P Architectural styles

Learning Unit Obligation

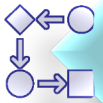
- Understand the Architecture style of MS
- Realise the design features and properties of MS
- Understand Choreography & Orchestration designs
- Understand the strength and weaknesses of MS
- Understand the deployment envirnoment of MS
- Realise how to mitigate MS shortcomings

Agenda


Warm up in Style


Peer-to-Peer Style

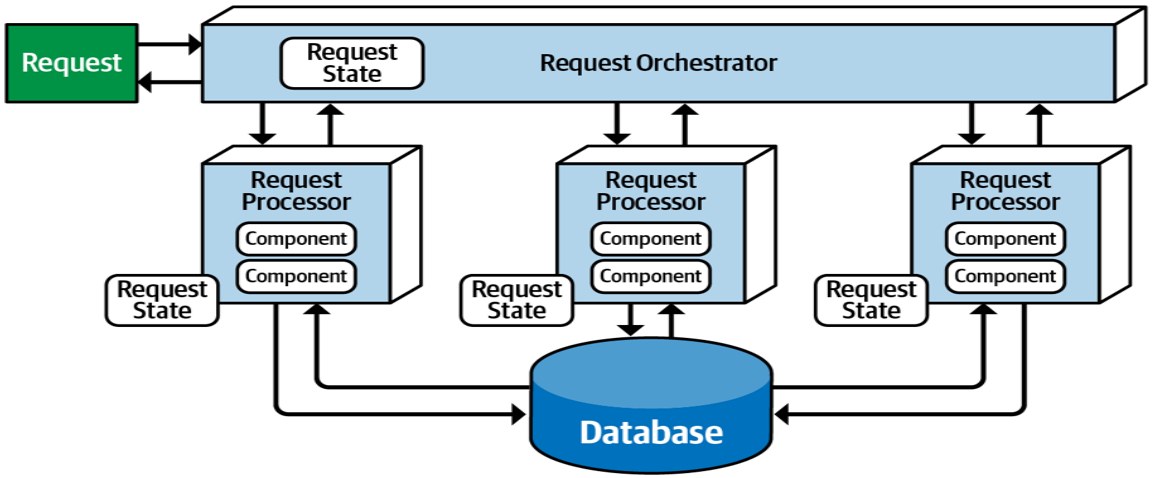

Microservices Style


Event-Driven Style


Choosing a Style

Event-Driven Architecture Style

- The *event-driven* architecture style is a popular distributed asynchronous architecture style used to produce highly scalable and high-performance applications.
- It is also highly adaptable and can be used for small applications and as well as large, complex ones.
- Event-driven architecture is made up of decoupled event processing components that asynchronously receive and process events.
- It can be used as a standalone architecture style or embedded within other architecture styles (such as an event-driven microservices architecture).
- Most applications follow what is called a *request-based* model as shown:



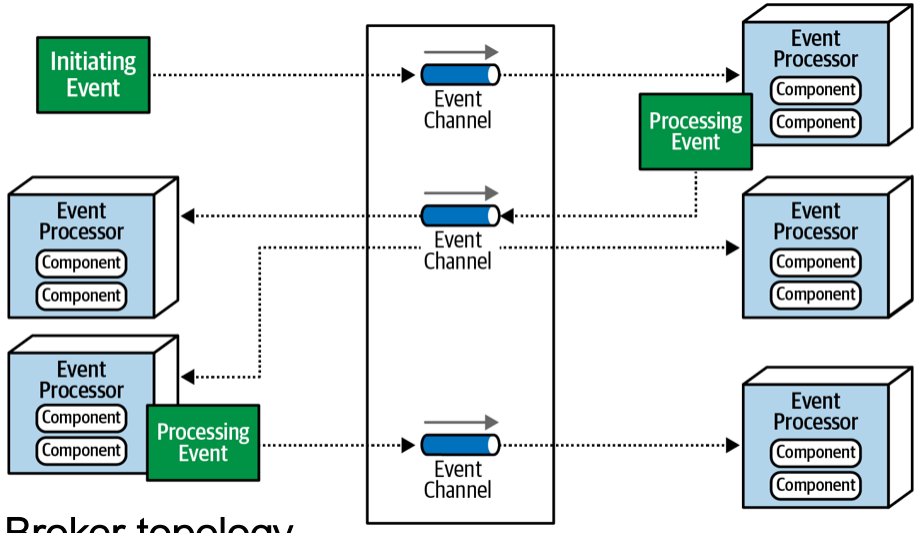
Event-Driven Architecture Style

Topology

There are two primary topologies within event-driven architecture:

Mediator topology: is commonly used when you require control over the workflow of an event process, we shall discuss this later.

Broker topology: is used when you require a high degree of responsiveness and dynamic control over the processing of an event. There is no central event mediator. The message flow is distributed across the event processor components in a chain-like broadcasting fashion through a lightweight message broker.



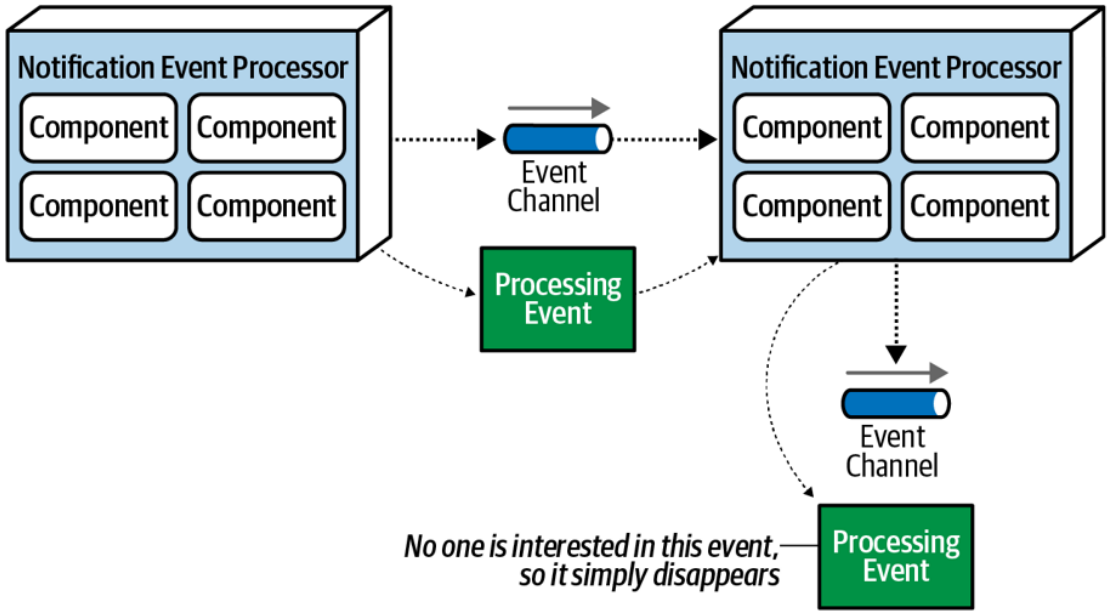
Broker topology

The event processor that accepted the initiating event performs a specific task associated with the processing of that event, then asynchronously advertises what it did to the rest of the system by creating what is called a *processing event*. This processing event is then asynchronously sent to the event broker for further processing, if needed.

Event-Driven Architecture Style

Broker topology

A good practice within the broker topology for each event processor to advertise what it did to the rest of the system, regardless of whether or not any other event processor cares about what that action was. This practice provides architectural extensibility if additional functionality is required for the processing of that event.



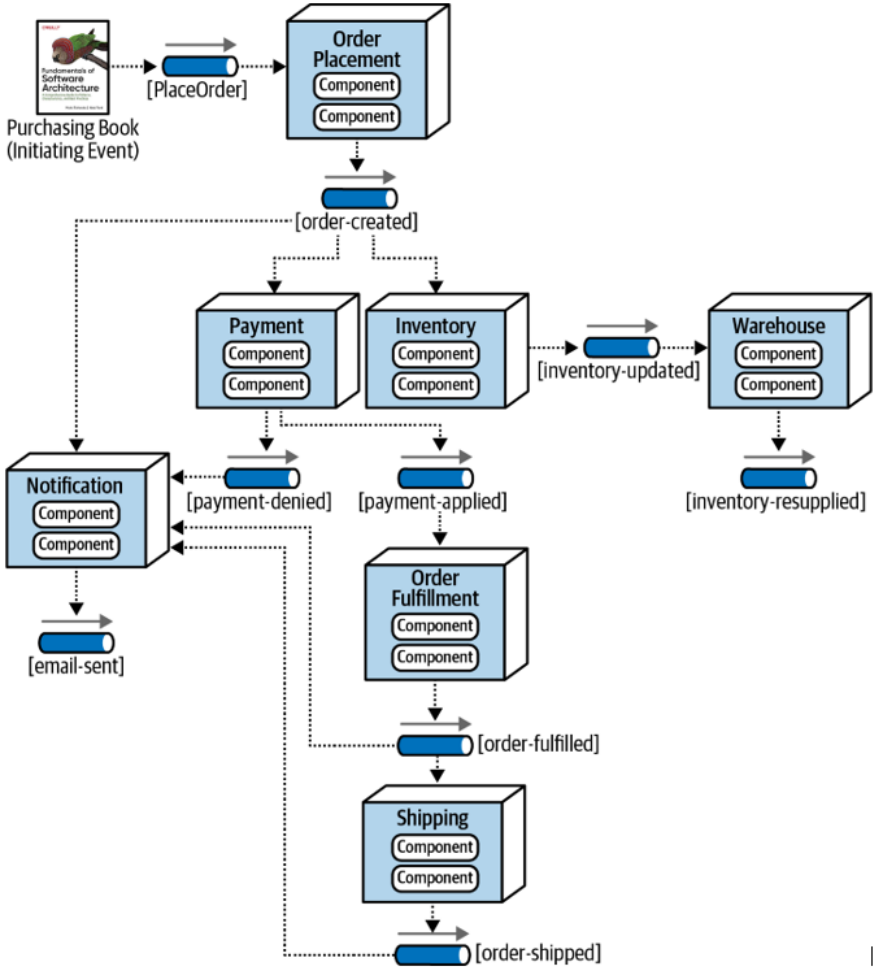
relay race & baton

Event-Driven Architecture Style

Broker topology

To illustrate how the broker topology works, consider the processing flow in a typical retail order system as it is placed for an item.

In this example, the OrderPlacement event processor receives the initiating event (PlaceOrder), inserts the order in a database table, and returns an order ID to the customer. It then advertises to the rest of the system that it created an order through an order-created processing event.



Event-Driven Architecture Style

Broker topology

While performance, responsiveness, and scalability are all great benefits of the broker topology, there are also some negatives things:

- There is no control over the overall workflow associated with the initiating event
- Error handling is also a big challenge with the broker topology without mediator.
- The business process is unable to move without automated or manual intervention.
- All other processes are moving along without regard for the error. For example, the Inventory event processor still decrements the inventory, and all other event processors react as though everything is fine.

Advantages	Disadvantages
Highly decoupled event processors	Workflow control
High scalability	Error handling
High responsiveness	Recoverability
High performance	Restart capabilities
High fault tolerance	Data inconsistency

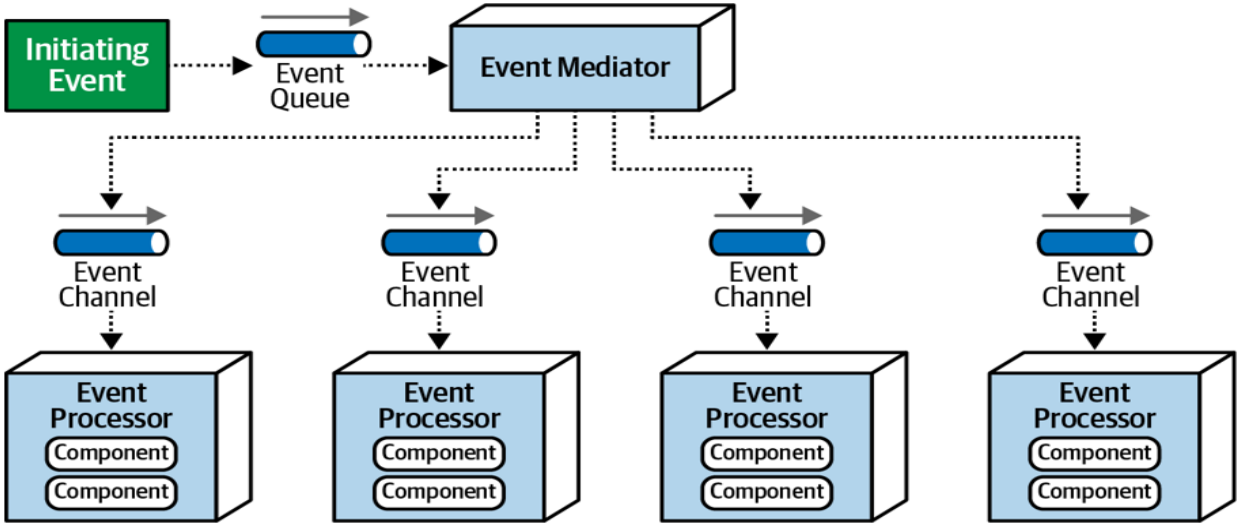
Event-Driven Architecture Style

Mediator Topology

The mediator topology of event-driven architecture addresses some of the shortcomings of the broker topology.

Central to this topology is an event mediator, which manages and controls the workflow for initiating events that require the coordination of multiple event processors.

The architecture components that make up the mediator topology are an initiating event, an event queue, an event mediator, event channels, and event processors.



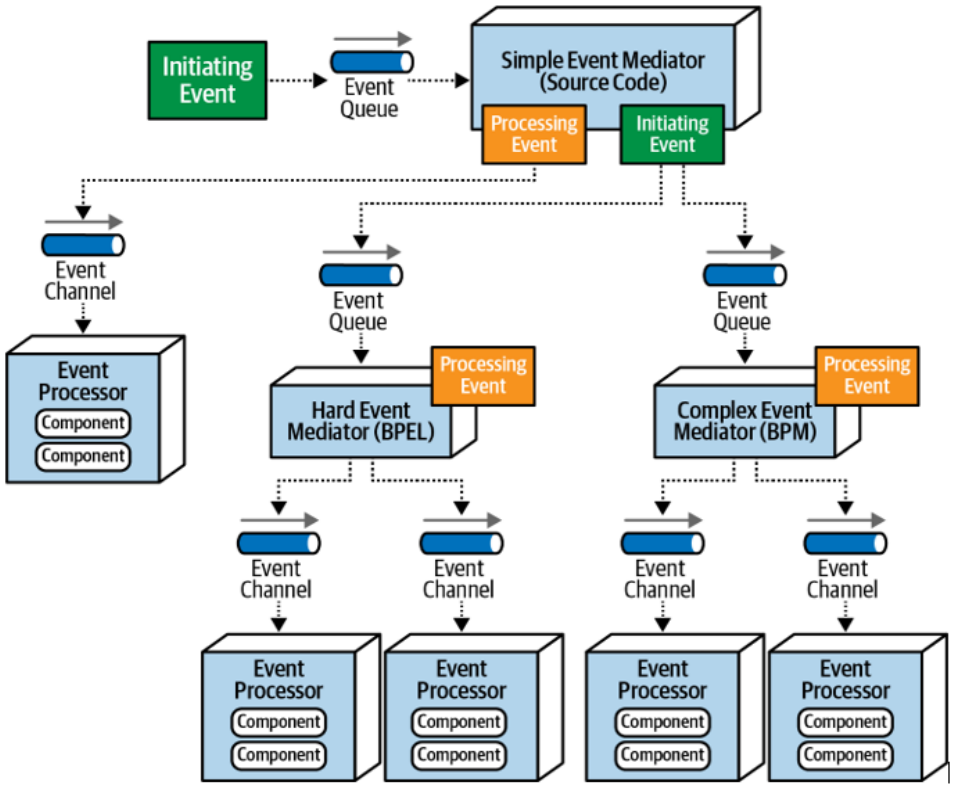
To reduces SPOF and also increases overall throughput and performance, Event Mediator are redundant.

Event-Driven Architecture Style

Mediator delegation model

We recommend classifying events as simple, hard, or complex and having every event always go through a simple mediator (such as Apache Camel or Mule). The simple mediator can then interrogate the classification of the event, and based on that classification, handle the event itself or forward it to another, more complex, event mediator.

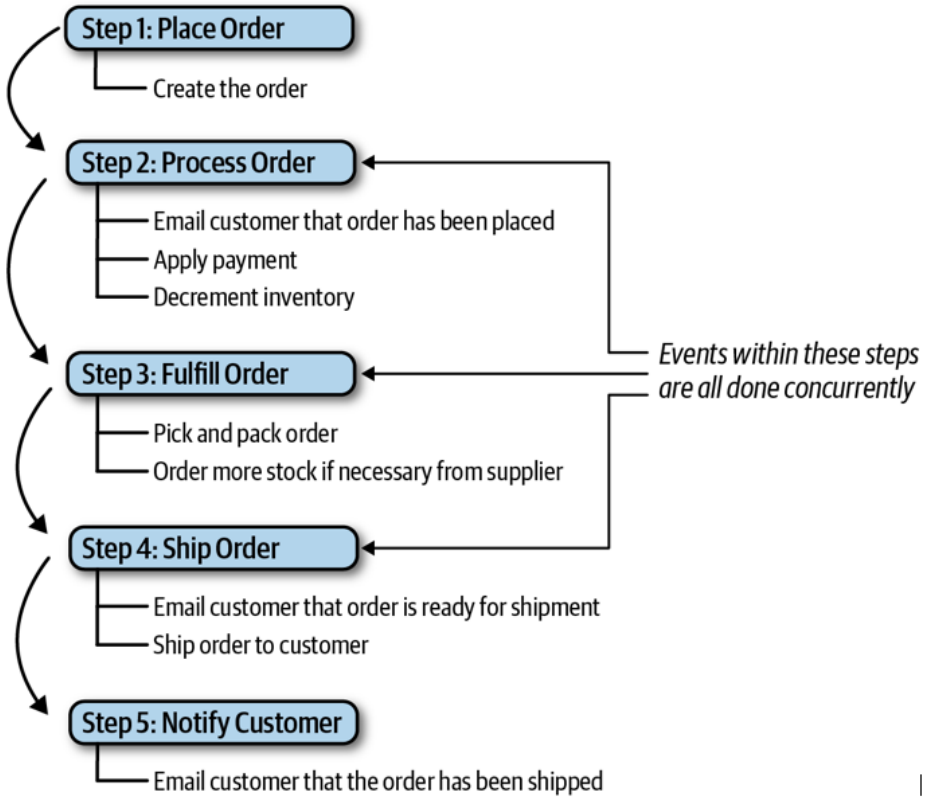
In this manner, all types of events can be effectively processed by the type of mediator needed for that event. This mediator delegation model is shown here.



Event-Driven Architecture Style

Mediator Topology

Notice that when the initiating event coming into the Simple Event Mediator is classified as either hard or complex, it forwards the original initiating event to the corresponding mediators (BPEL or BPM). To illustrate how the mediator topology works, consider the same retail order entry system example described in the prior broker topology section, but this time using the mediator topology.



Event-Driven Architecture Style

Mediator Topology

A hybrid model combining both the mediator and broker topologies can be used to address the dynamic nature of complex event processing.

The table below shows the trade-offs for the mediator topology:

Advantages	Disadvantages
Workflow control	More coupling of event processors
Error handling	Lower scalability
Recoverability	Lower performance
Restart capabilities	Lower fault tolerance
Better data consistency	Modeling complex workflows

Event-Driven Architecture Style

Event-Driven style implementation considerations

MS style Advantages	MS style Disadvantages
High scalability	Testability challenge
High performance	Can get complex
High Fault tolerance	High overhead for security
Very high modularity	Hybrid tactics needed
Highly evolutionary	Complex workflow

Event-Driven Architecture Style

Learning Unit Obligation

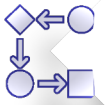
- Understand the two styles of ED architecture
- Realise the design features and properties of ED style
- Understand Broker and Mediator ED strength and weakness
- Understand the difference in handling simple and complex events
- Understand the ED implementation considerations

Agenda

Warm up in Style

Peer-to-Peer Style

Microservices Style



Event-Driven Style

Choosing a Style

Choosing a Style

Choosing an architecture style represents the culmination of analysis about:

- Trade-offs for architecture characteristics (attributes)
- Domain considerations
- Strategic goals
- Optimization tactics,
- Business and market considerations
- Agility and CD considerations

However contextual the decision is, some general advice exists around choosing an appropriate architecture style.

Choosing a Style

Decision Criteria

Architects should go into the design decision with the following things:

The domain (analysis & implementation)

Domain affects operational architecture attributes. Architects must have at least a good general understanding of the major aspects of the domain features under design.

Attributes that impact Architecture

Architects must discover the architecture attributes needed to support the domain and other external factors.

Data architecture

Architects and DBAs must collaborate on database, schema, and other data-related design concerns.

Choosing a Style

Decision Criteria

Organizational factors

Many external factors may influence design. For example, the cost of a particular cloud vendor, and TTM may prevent the ideal design.

Knowledge of process, teams, and operational concerns

Many specific project factors influence an architect's design, such as the software development process, interaction (or lack of) with operations, and the QA process.

For instance, an insurance company application consisting of multipage forms, each of which is based on the context of previous pages, would be difficult to model in microservices.

Choosing a Style

Decision Criteria

Monolith versus distributed

A single set implies that a monolith is suitable, whereas different architecture characteristics imply a distributed architecture.

Where should data live?

If the architecture is monolithic, architects commonly assume a single relational databases or a few of them. In a distributed architecture, the architect must decide which services should persist data, which also implies thinking about how data must flow throughout the architecture to build workflows.

Choosing a Style

Decision Criteria

What communication styles between services—synchronous or asynchronous?

Once the architect has determined data partitioning, their next design consideration is the communication between services—synchronous or asynchronous?

Synchronous communication is more convenient in most cases, but it can lead to scalability, reliability, and other undesirable characteristics.

Asynchronous communication can provide unique benefits in terms of performance and scale but can present a host of headaches: data synchronization, deadlocks, race conditions, debugging, and so on.

Architects should default to synchronous when possible and use asynchronous when necessary.

Choosing a Style

Distributed Case Study: Going, Going, Gone

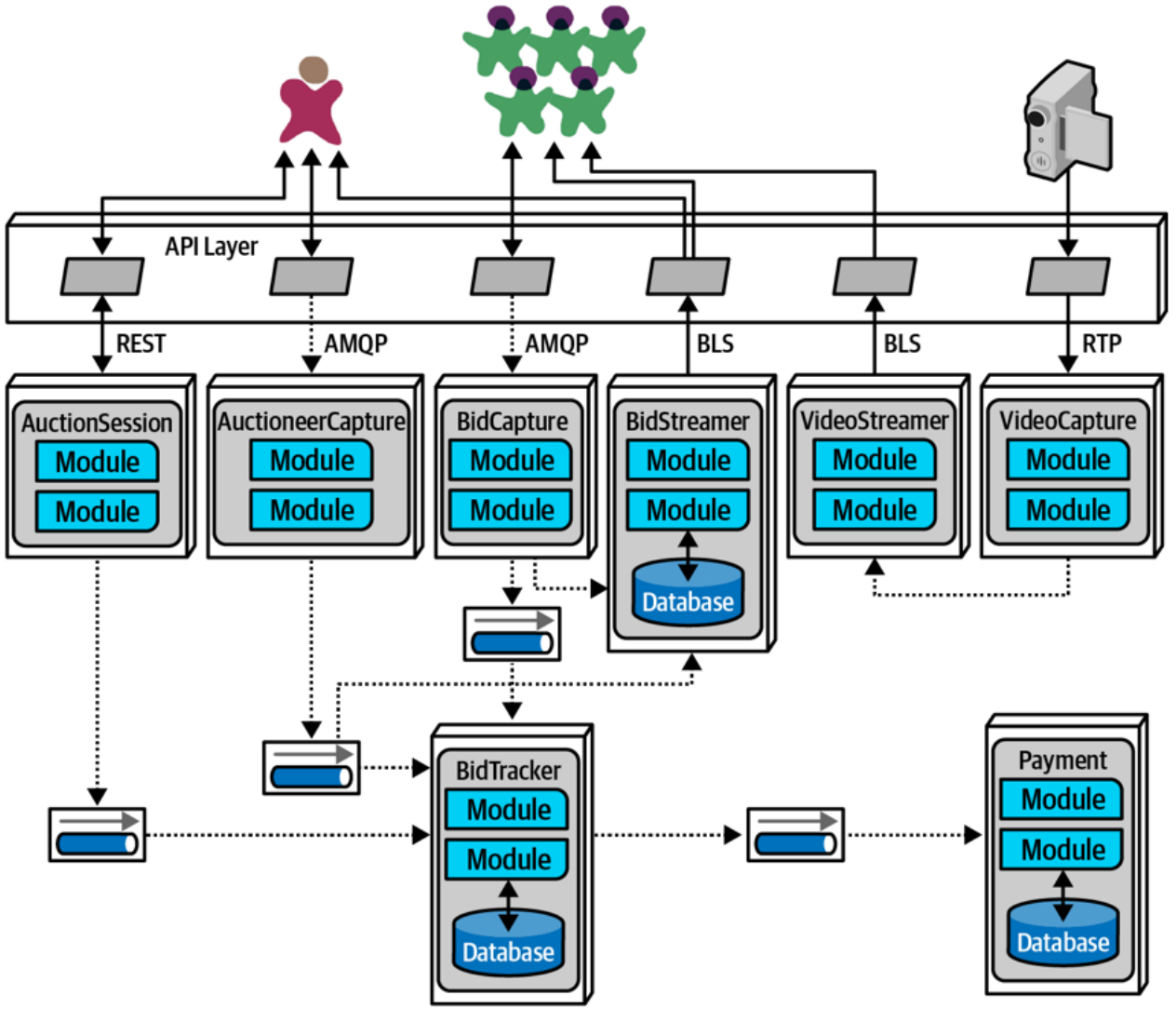
The requirements for GGG also explicitly state certain ambitious levels of scalability, elasticity, performance, and a host of other tricky operational architecture characteristics.

Of the candidate distributed architectures, either low-level event-driven or microservices match most of the architecture characteristics.

Achieving the stated performance will provide a challenge in microservices, but architects can often address any weak point of an architecture by designing to accommodate it. For example, while microservices offers a high degrees of scalability naturally, architects commonly have to address specific performance issues caused by too much orchestration, too aggressive data separation, and so on.

Choosing a Style

microservices implementation of Going, Going, Gone



Choosing a Style

microservices implementation of Going, Going, Gone

Each identified component became services in the architecture, matching component and service granularity. GGG has three distinct user interfaces:

Bidder

The numerous bidders for the online auction.

Auctioneer

One per auction.

Streamer

Service responsible for streaming video and bid stream to the bidders. Note that this is a read-only stream.

Choosing a Style

microservices implementation of Going, Going, Gone

The following services appear in this design of the GGG architecture:

BidCapture

Captures online bidder entries and asynchronously sends them to Bid Tracker. This service needs no persistence because it acts as a conduit for the online bids.

BidStreamer

Streams the bids back to online participants in a high performance, read-only stream.

BidTracker

Tracks bids from both Auctioneer Capture and Bid Capture. This is the component that unifies the two different information streams, ordering the bids as close to real time as possible. Note that both inbound connections to this service are asynchronous.

Choosing a Style

microservices implementation of Going, Going, Gone

Auctioneer Capture

Captures bids for the auctioneer.

Auction Session

This manages the workflow of individual auctions.

Video Streamer

Streams the auction video to online bidders.

Choosing microservices, then intelligently using events and messages, allows the architecture to leverage the most out of a generic architecture pattern while still building a foundation for future development and expansion.

Choosing a Style

Learning Unit Obligation

- Understand design analysis for selecting an architectural style
- Identify the decision criteria to select the best fit architecture style
- Understand the GGG architectural design drivers