Fundamentals of Software Architecture
DIT344

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

System Requirement

What is a requirement?

Definition:

a requirement is a statement (Scope of Work SOW) describing:

1. *an aspect of what the system functions must do,*
2. *a constraint on the system's development*
3. *adequately solving the stakeholders' problem*

System Requirement

The functional requirements: WHAT as follows:

- What I/O the system should expect
- What data structure and medium to use.
- What computations the system should perform
- What is the User and system admin functions

Non-Functional Requirement

The attributes “HOW”, to be covered later

System Requirement

Constraints

Constraints a set restrictions on how the user requirements are to be implemented.

- Interface APIs Requirements.
- Communication (protocols) Interfaces.
- Hardware Interfaces.
- Software compatibility & Interfaces.
- User Interfaces & experience
- Language, code, and reusability
- Testing and maintenance

System Requirement

Agile approaches to requirements

You do not develop large requirements documents. Instead, two approaches are employed: user stories and test case.

User story: is similar to a use case, but has a looser structure; it describes some proposed software feature from the perspective of how the user will use it and should be limited to about three sentences. Development proceeds by choosing a very small number of user stories to implement in the next iteration. Ideally each iteration will take only a few days to develop.

Test case: The first stage of development in many agile approaches is to first develop test cases. The series of test cases becomes the detailed specification of how a user story should be implemented.

Use Cases Brief

Use Case (how actors will use the system)

- Determine the types of users or systems that will use the system.
- It is a typical sequence of actions that an actor performs in order to complete a given task.
- An actor is a role that a user or some other system plays when interacting with system.
- Most of the actors will be users; a given user may be considered as several different actors
- A use case should include only actions in which the actor interacts with the system.

Use Cases Brief

How to describe a single use case

1. **Name**. Give a short, descriptive name to the use case.
2. **Actors**. List the actor or actors who can perform this use case.
3. **Goals**. Explain what the actor or actors are trying to achieve.
4. **Preconditions**. Describe the state of the system before the use case occurs.
5. **Summary**. Summarize what occurs as the actor or actors perform the use case.
6. **Related use cases** . List use cases that may be generalizations, specializations, extensions or inclusions of this one.
7. **Steps** . Describe each step of the use case using a two-column format.
8. **Postconditions** . What state is the system in following the completion of the use case.

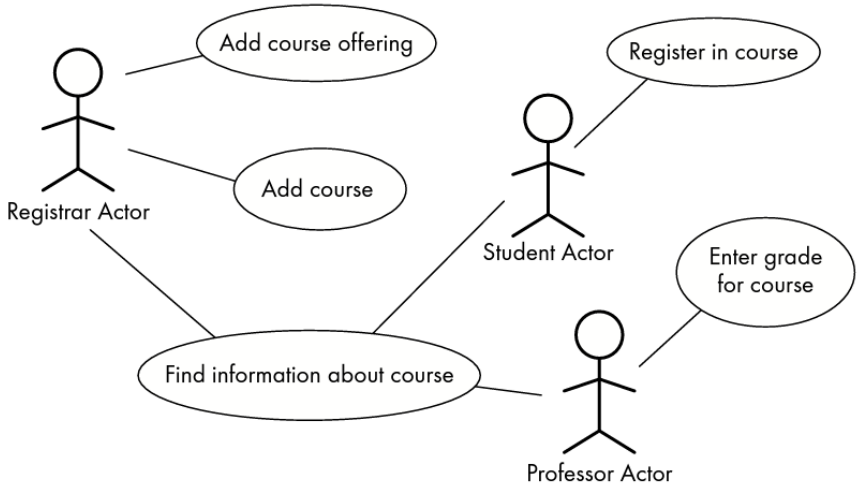
Use Cases Brief

Use case diagrams

Use case diagrams are UML's notation for showing the relationships among a set of use cases and actors.

They help a software engineer to convey a high-level picture of the functionality of a system.

There are two main symbols in use case diagrams: an actor is shown as a stick person and a use case is shown as an ellipse. Lines indicate which actors perform which use cases.



A simple use case diagram showing three actors and five use cases

Use Cases Brief

The use case modeler can use extensions , generalizations or inclusions to represent different types of relationships among use cases.

Extensions are used to make optional interactions or handle exceptional cases.

Generalizations use triangle symbol: several similar use cases can be shown along with a common generalized use case. Same like parent and child.

Inclusions allow you to express a part of a use case so that you can capture commonality between several different use cases.

Use Cases Brief

Scenarios

A scenario is an instance of a use case

It can help to clarify the associated use case.

It is also often simply a use case instance .

Example: Describe a concrete scenario corresponding to the 'Exit car park, paying cash' use case from Example 4.11.

Steps:

Actor actions	*System responses*
Drives to the exit barrier.	Detects the presence of a car.
	Displays: 'Please insert your ticket'.
Inserts ticket.	Displays: 'Amount due $2.50'.
Inserts $1 into the slot.	Displays: 'Amount due $1.50'.
Inserts $1 into the slot.	Displays: 'Amount due $0.50'.
Inserts $1 into the slot.	Returns $0.50.
	Displays: 'Please take your $0. 50 change'.
	Raises barrier.
Drives through barrier, triggering sensor.	Lowers barrier.

Quality Attributes

Systems are frequently redesigned not because they are functionally deficient.

Stakeholders decide value and priorities of functions and attribute.

It is the mapping of a system's functionality onto software architecture that determines the architecture's quality attributes.

A quality requirement is a specification of the acceptable quality attribute.

A quality attribute is a measurable or testable property of a system.

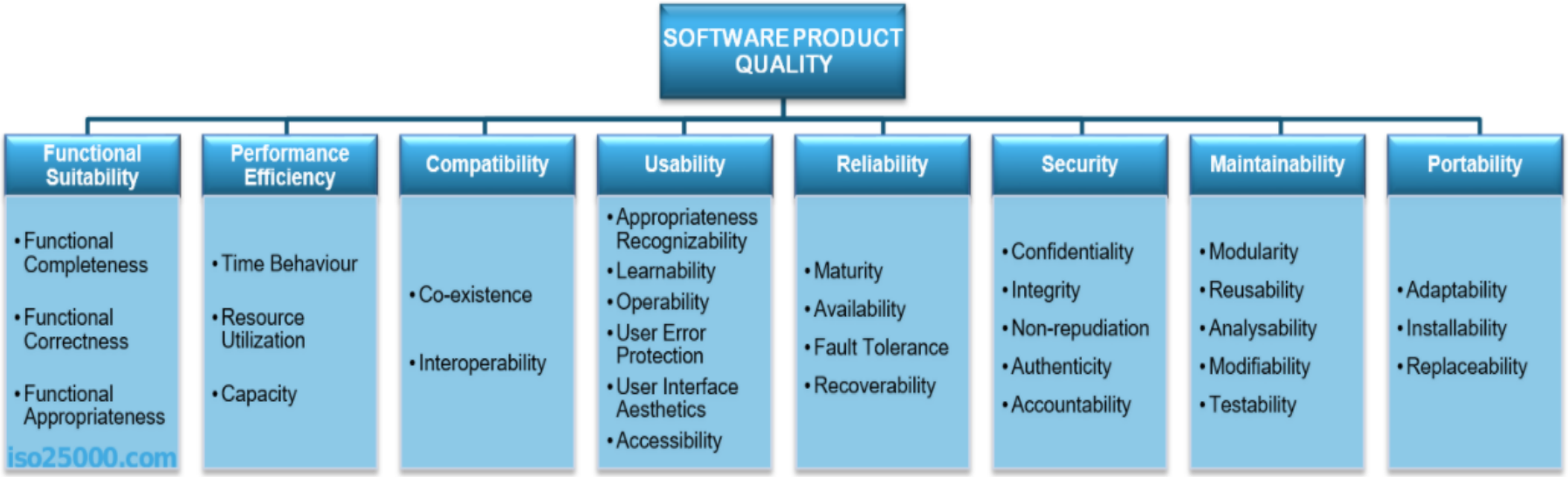
Quality attributes should be communicated based on KPIs that are agreeable across all stakeholders.

Quality Attributes

ISO/IEC 25010 Quality Model*

The quality of a system is the degree to which the system satisfies the stated and implied needs of its various stakeholders.

The product quality model defined in ISO/IEC 25010 comprises the eight quality characteristics shown in the following figure:



*https://iso25000.com/index.php/en/iso-25000-standards/iso-25010#:~:text=ISO%2FIEC%2025010&text=The%20quality%20model%20determines%20which,stakeholders%2C%20and%20thus%20provides%20value.

Quality Attributes

Availability

Availability is the percentage of time when system it is operational.

$$A = \frac{MTBF}{MTBF + MTTR}$$

Mean Time Between Failures (MTBF)

Number of hours that pass before a component fails

E.g. 2 failures per million hours: MTBF = 10^6 / 2 = 0,5 * 10^6 hr

Availability	Downtime
90% (1-nine)	36.5 days/year
99% (2-nines)	3.65 days/year
99.9% (3-nines)	8.76 hours/year
99.99% (4-nines)	52 minutes/year
99.999% (5-nines)	5 minutes/year
99.9999% (6-nines)	31 seconds/year !

Calculate **MTTR** by dividing the total **time** spent on unplanned maintenance by the number of **times** an asset has failed

Quality Attributes

Evaluating Quality Attributes

Quality attributes can be evaluated through:

- **Scenario-based evaluation**: eg. *scenarios* for assessing maintainability
- **Simulation**: a part of the architecture is implemented and executed in the actual system context.
- **Test Environment**: Controlled nonproduction testing with similar environment
- **Mathematical modeling**: checking for *potential deadlocks, and performance.*.
- **Prototype or emulated concept design:** Build a concept system, **MVP** (suitable for large systems)

Quality Attributes

Evaluating Quality Attributes

Overlapping concerns

Performance: due to DDS attack or poor design

Security: due to poor layering or internal compromise

Untestable Concern

The quality attribute should be tested in all the circumstances. (stress condition)

Gathering Quality Attribute Information

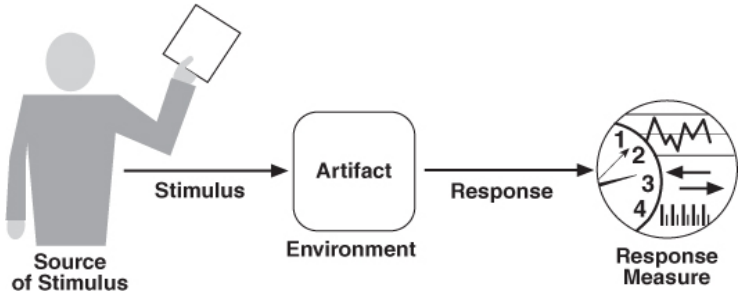
Quality requirements and design constraints are enabled by two main techniques:

- Quality Attribute Scenario (QAS) and
- Quality Attribute Workshop (QAW).

QA Scenarios

We specify quality attribute requirements, we capture them formally as six parts of QAS:

1. *Source of stimulus*. (a human, or any other actuator) that generated the stimulus.
2. *Stimulus.* A condition that requires a response. For different quality it means something specific.
3. *Environment*. The system may be in an overload condition, test, or in normal operation.
4. *Artifact.* Some artifact is stimulated. This may be a collection or whole system, or pieces of it.
5. *Response.* The response is the activity undertaken as the result of the arrival of the stimulus.
6. *Response measure.* A response should be measurable so that the requirement can be tested.



Parts of a quality attribute scenario (ex. web portal responsiveness).

QA Scenarios

There are two types of QAS: general and concrete:
- A General scenario do not belong to any system.
- A Concrete scenario belongs to a particular system under specific conditions.

Table 3.1 General QAS for availability quality

Source	The source can be internal or external, for example, people, hardware, software, physical infrastructure, etc.
Stimulus	Fault
Artifacts	Processors, communication channels, persistent storage, process
Environment	Normal operation, shutdown, repair mode, overloaded operation
Response	Prevent fault from becoming failure Detect the fault Recovery from the fault
Response measure	Availability percentage, for example (99.999%), time to detect fault, time to repair fault, time or time interval in which a system can be in a degraded mode, etc.

Table 3.3 Concrete table

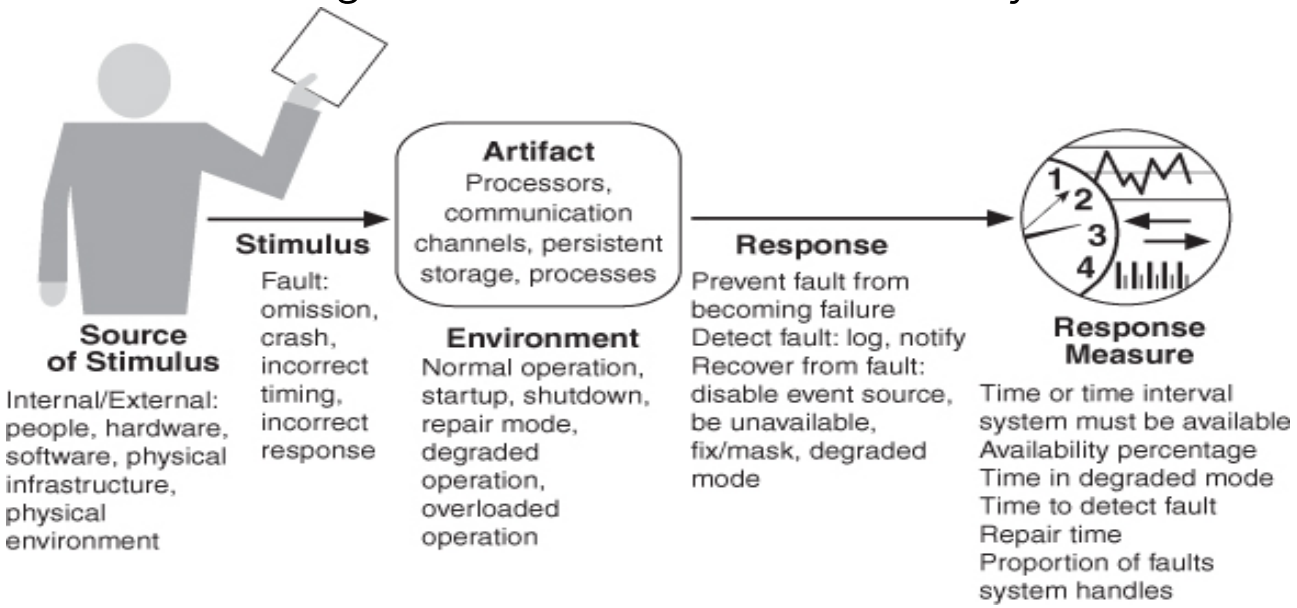
Source	Internal hardware
Stimulus	Crash
Artifacts	Processors
Environment	Normal operation
Response	Detect the fault Recovery from the fault
Response measure	System can be in a degraded mode no more than 15 minutes

QA Scenarios

We develop first the **general quality attribute scenarios**, for a specific attribute such as availability.

Then we translate them to the specific requirement of the system under development to get concrete scenarios, by specifying the source and the stimulus.

A general scenario for availability



Architectural Styles
Part III

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

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?

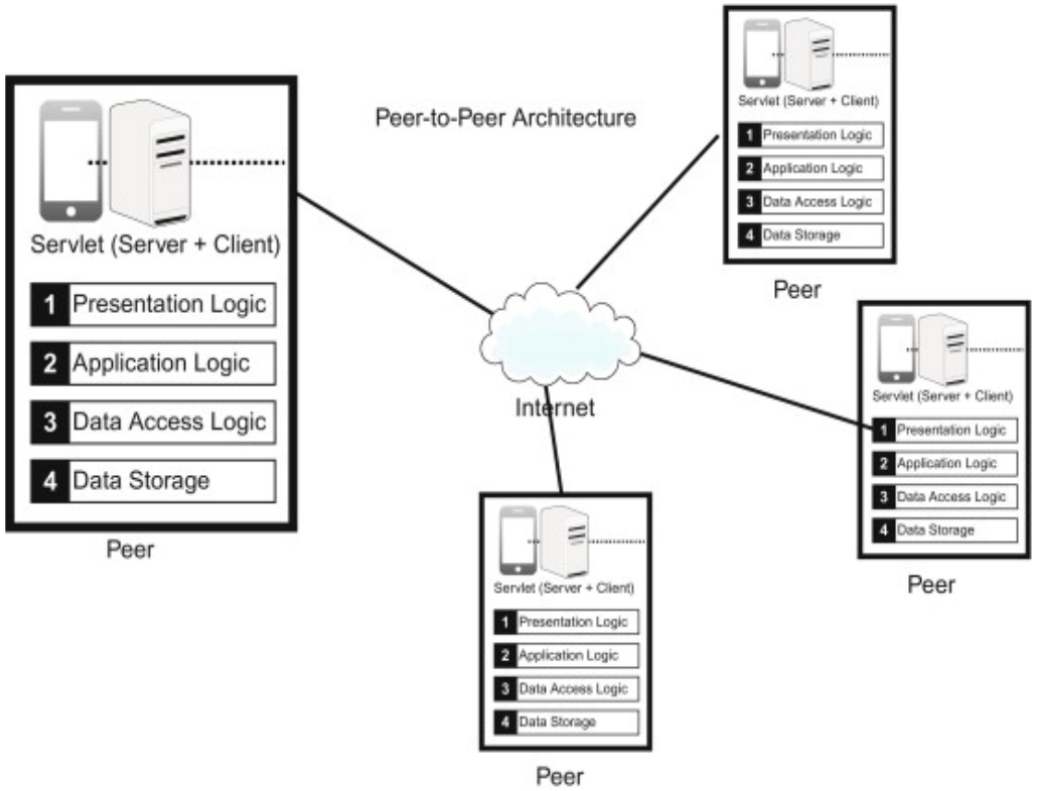
It is a type of computing architecture that enables computing components or layers to execute independently while still interfacing with each other.

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.

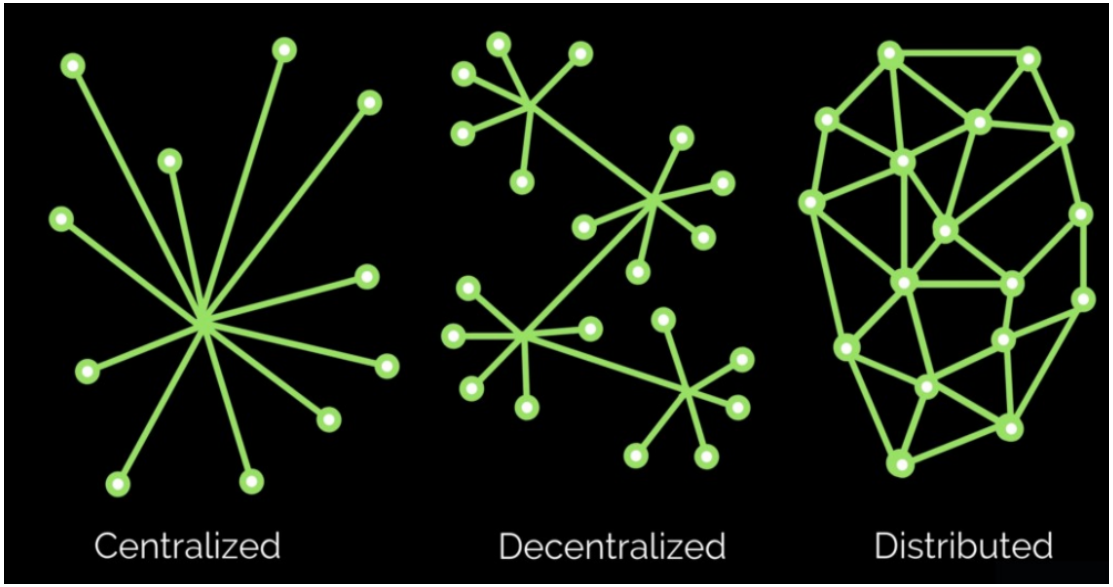
Decoupled architecture helps achieve higher computing performance, deployment, reusability, and testability by isolating and executing individual components independently and in parallel.

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.

P2P Implementations

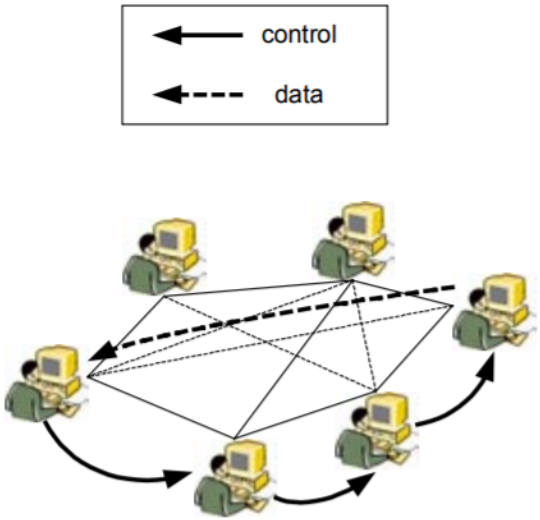
- Windows 10 updates are delivered both from Microsoft's servers and through P2P (bandwidth sharing)
- The decentralized framework of P2P systems makes them highly available and resistant to cyber attacks and also more scalable.
- The more users join it, the more resilient and scalable it gets. Bigger 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 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.



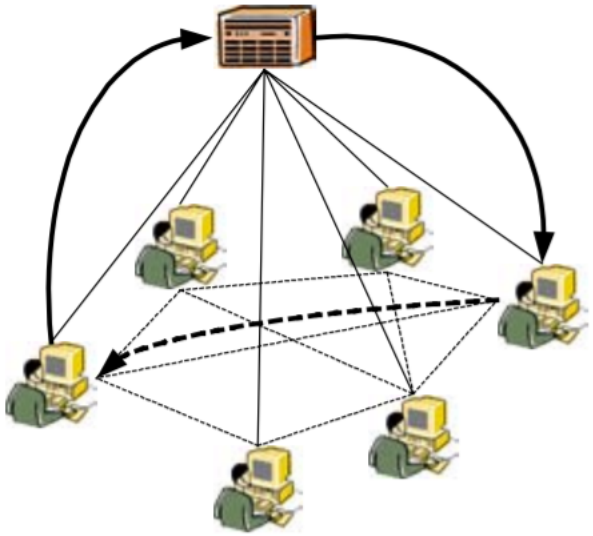
P2P Architectural styles

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.



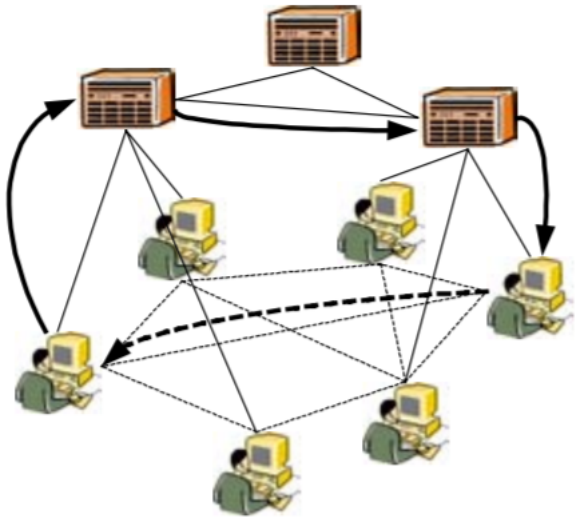
P2P Architectural styles

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

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

Microservices Style

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

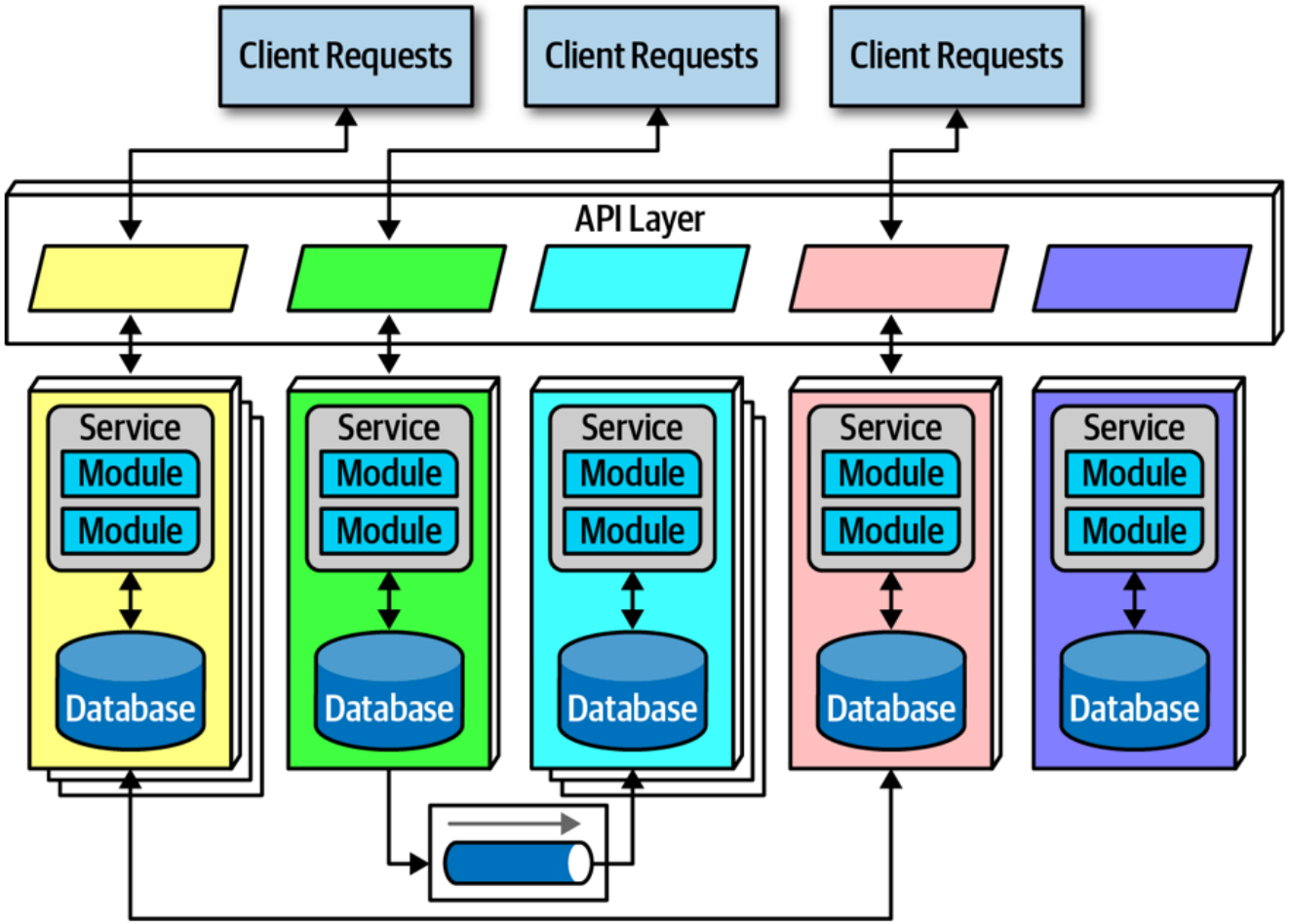


Figure 17-1. The topology of the microservices architecture style

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.

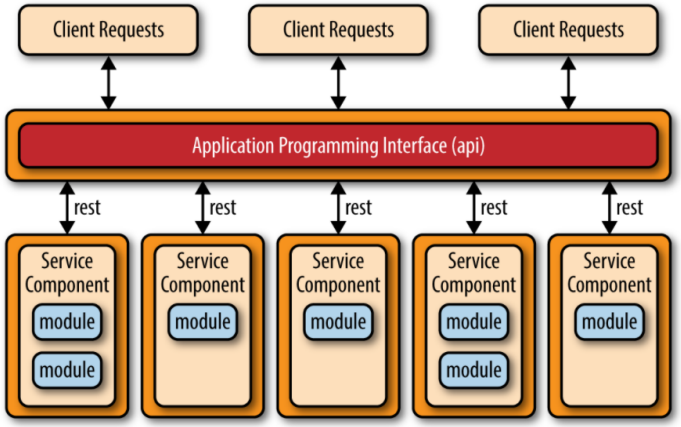
Bounded Context. The driving philosophy of microservices is the notion of *bounded context*: each service models a domain or workflow. Thus, each service includes everything necessary to operate within the application, including classes, other subcomponents, and database schemas.

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

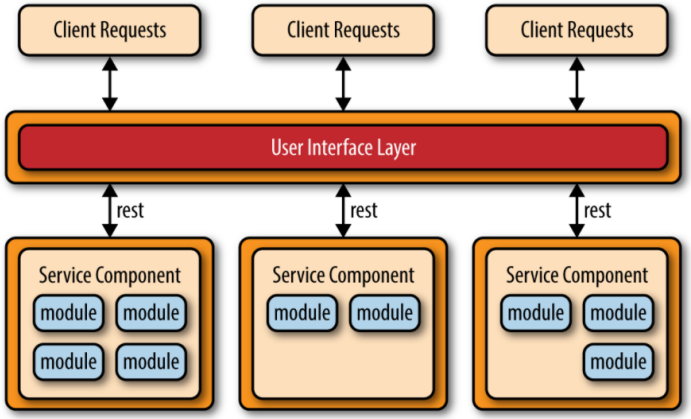
Microservices Style-Interface

The User Interface is critical to the performance of Microservice design.

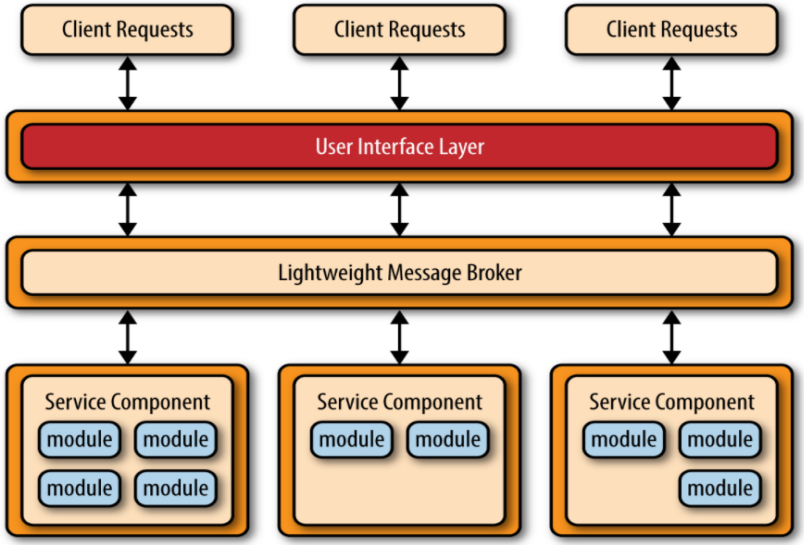
There are 3 different designs to deal with the APIs. Mainly this is suitable for remote web application environment.



Traditional web-based or fat-client business application



Web application remotely accesses separately deployed service.

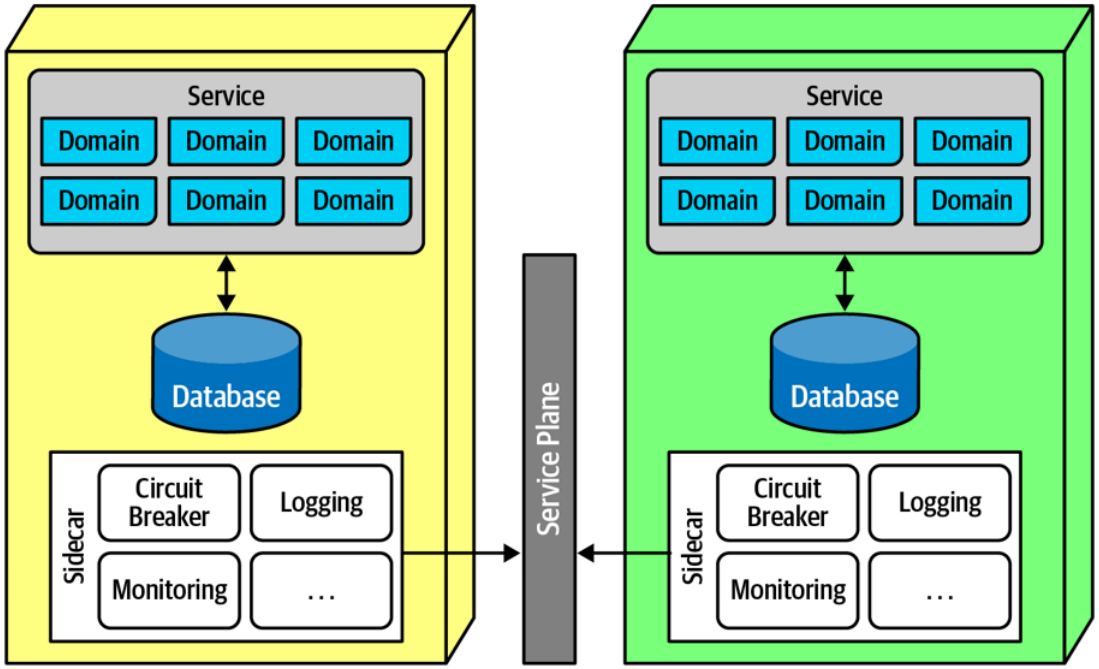


lightweight centralized message broker to access remote service components

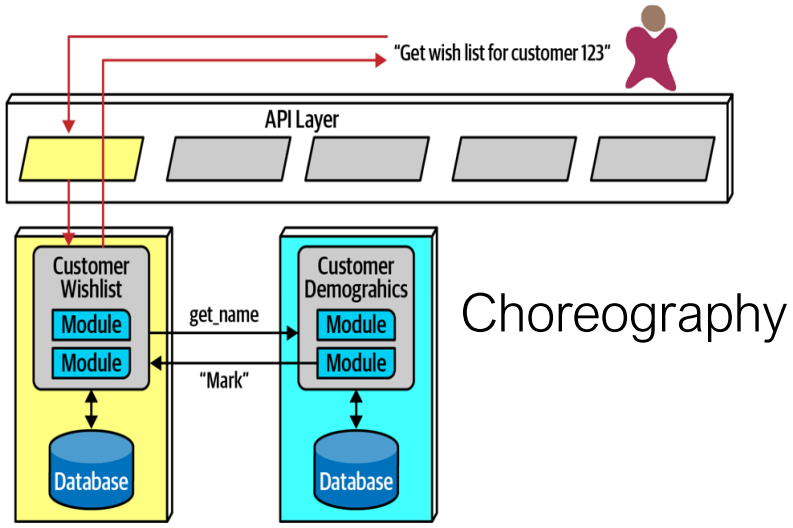
Rest: Remote access protocol

Microservices Style-Sidecar

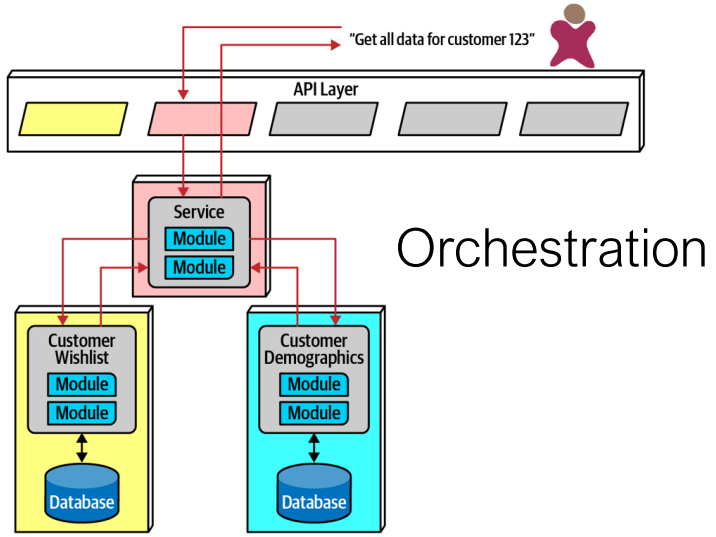
When a common operational concerns appear within each service as a separate component, the sidecar component can handle all the operational concerns that teams benefit from coupling together. Thus, when it comes time to upgrade the monitoring tool, the shared infrastructure team can update the sidecar, 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

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.

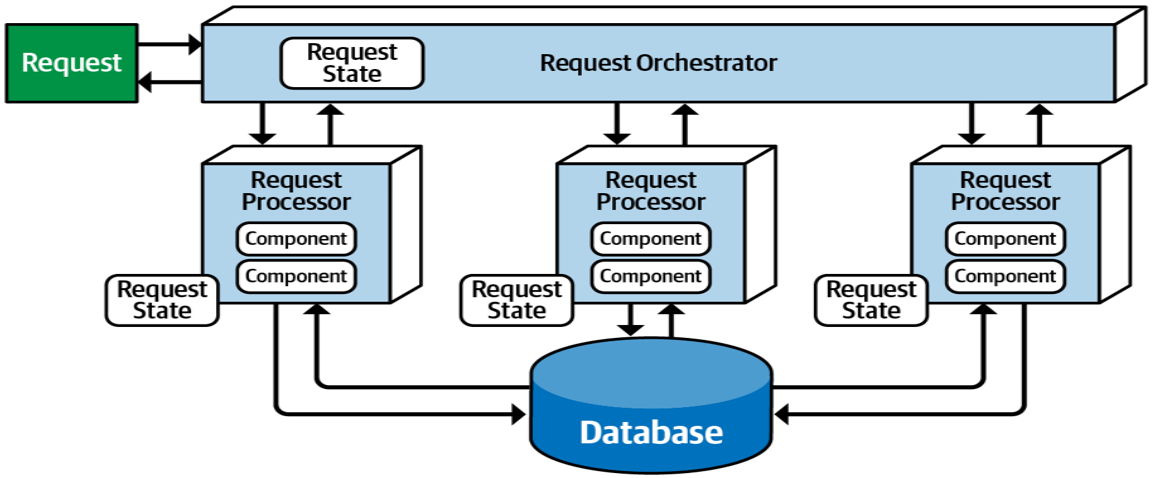
Microservices Style

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	

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:



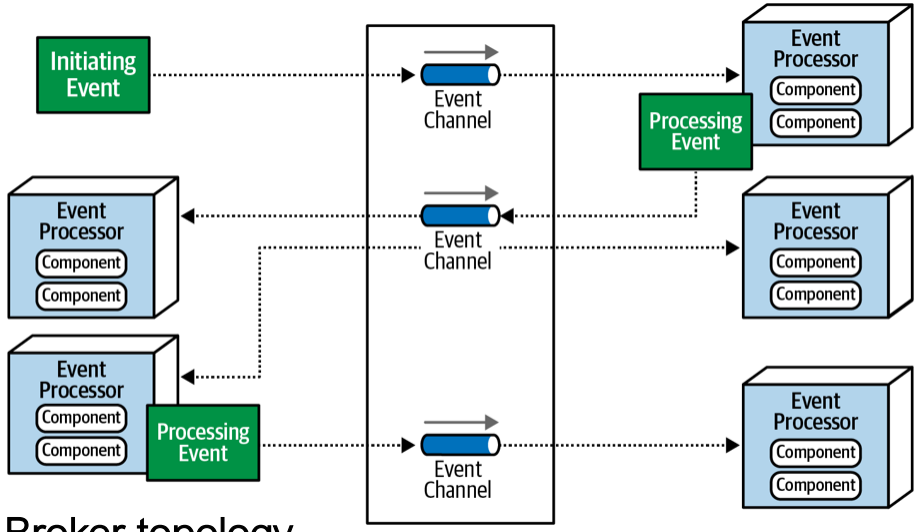
Event-Driven Architecture Style

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.



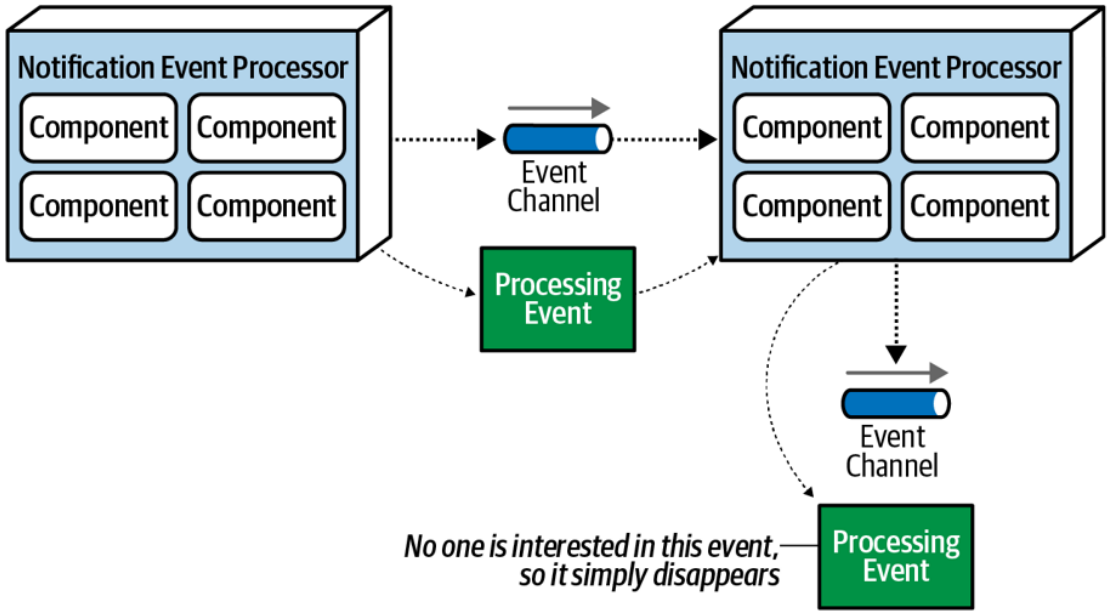
Broker topology

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.



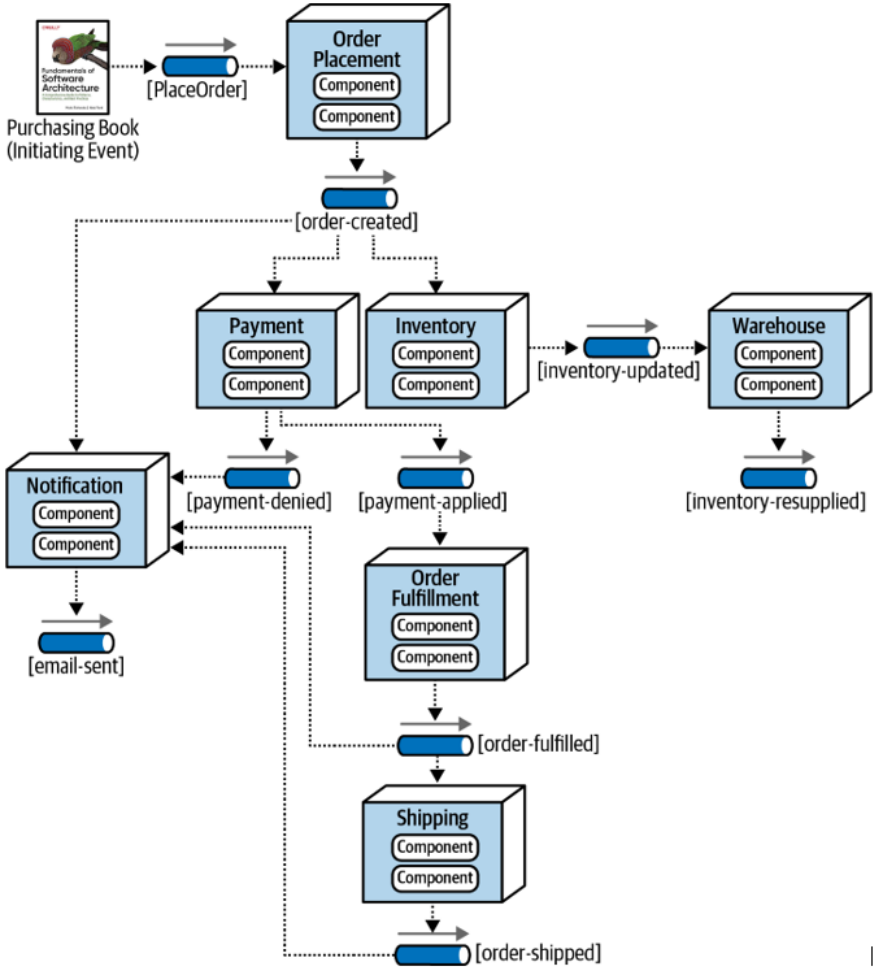
relay race & baton

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

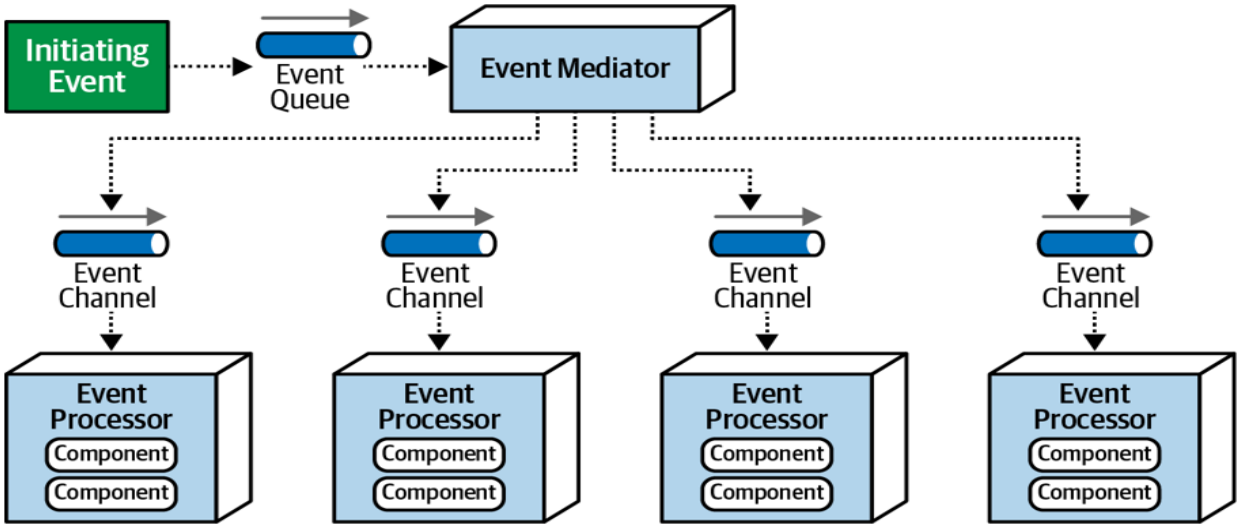
Event-Driven Architecture Style

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.



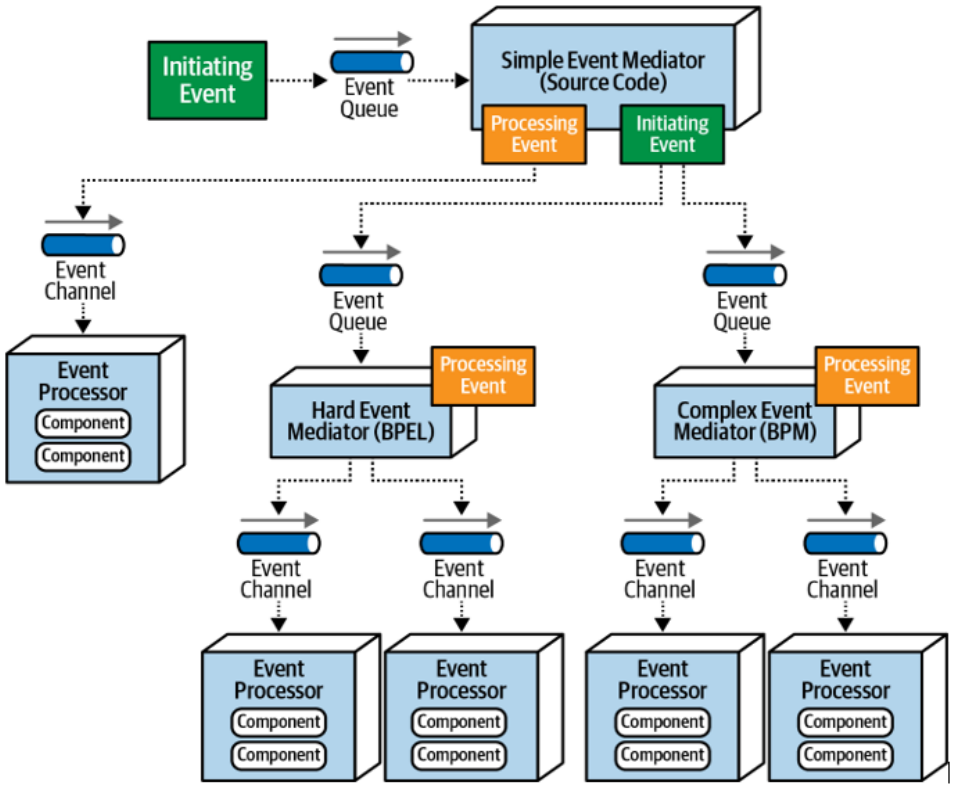
To reduces SPOF and also increases overall throughput and performance, Event Mediator are redundant.

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

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

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 business 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.

Choose an Architectural Style

The 5-Keys process

1- Key Use Cases or Scenarios (identify main scenarios)

2- Key Stakeholders requirement (must have requirements)

3- Key Attributes (must have attributes)

4- Key Style drivers (identify candidate styles)

5- Key Advantages/Dis-advantages (of candidate style)

If step 5 revealed high-impact disadvantages, then repeat process.

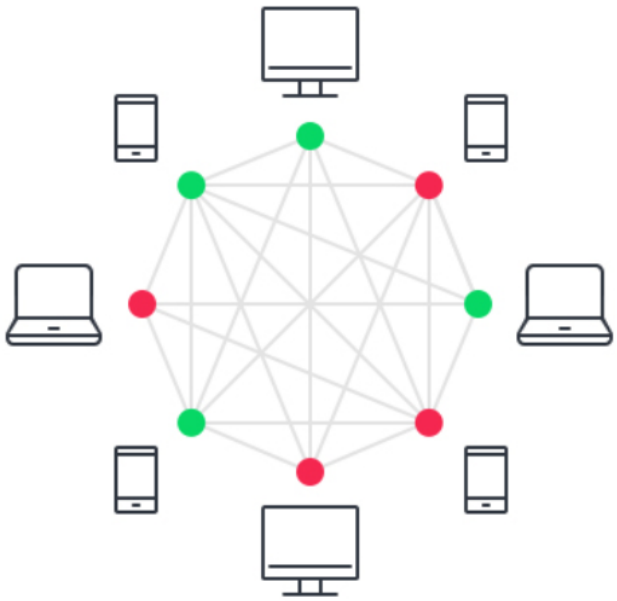
Blockchain Terminology

Types of Nodes

Blockchain nodes are not the same, Validators are able to validate, and issue new blocks (miners) with trustless consensus, while other basic nodes (users) who can initiate transactions.

Validator Node

Can both initiate/receive and validate transactions



Memeber Node

Can only initiate/receive transactions

Blockchain Terminology

Properties

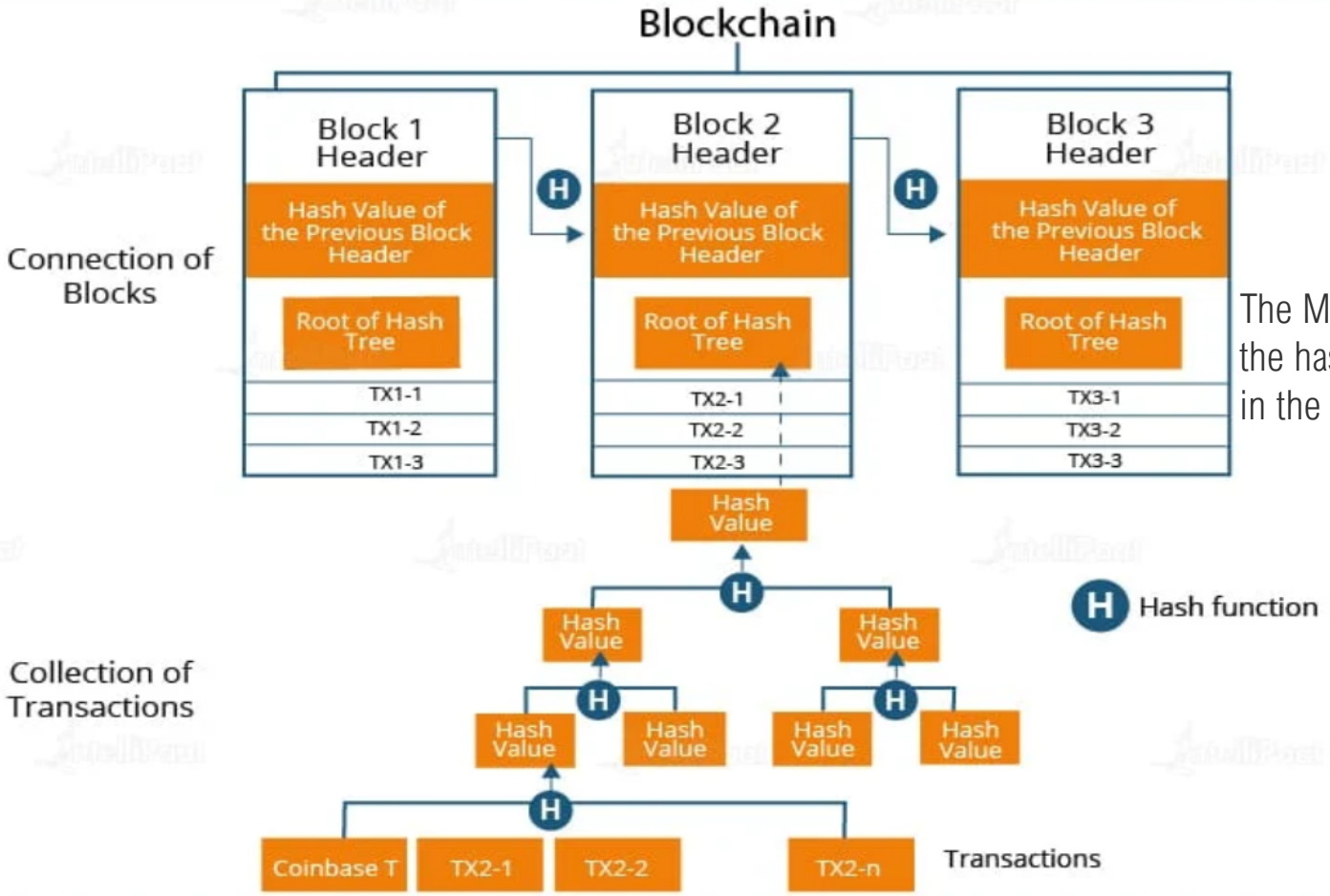
The following table provides a detailed comparison among these three blockchain systems:

Property	Public blockchain	Consortium blockchain	Private blockchain
Consensus determination	All miners	Selected set of nodes	Within one organization
Read permission	Public	Public or restricted	Public or restricted
Immutability level	Almost impossible to tamper	Could be tampered	Could be tampered
Efficiency (use of resources)	Low	High	High
Centralization	No	Partial	Yes
Consensus process	Permissionless	Needs permission	Needs permission

Blockchain Terminology

Blocks Structure

Each blockchain block consists of certain data, the hash of the block, the hash from the previous block, and some Transactions.



The Merkle root is the hash of all the hashes of all the transactions in the block.

Blockchain Architecture

What is the architecture behind the blockchain?

Blockchain is not just a distributed database; it includes advanced software and security techniques to create a network of nodes (peers) that are always in sync.

Each node validates and verifies transactions and blocks redundantly, in order to reach consensus, and it provides a platform to run decentralized applications.

To achieve this, the blockchain or digital ledger technology is built upon a layered architecture.

In most cases, this contains four or five layers, namely the data layer, network layer, consensus layer, incentive layer, and application layer.

Blockchain Architecture

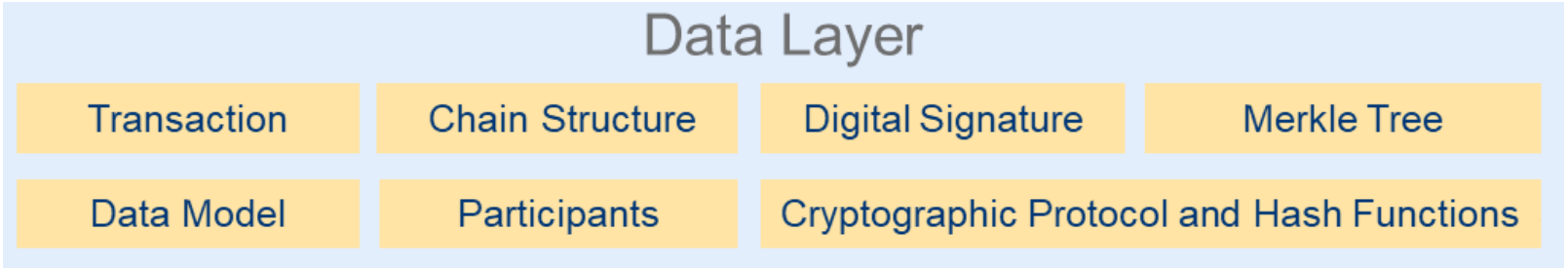
The data layer

At the bottom layer of the stack is the data layer, which deals with the data structure and the physical storage of data in the blockchain.

The diagram below shows the common capabilities that are part of this layer:

The capabilities of the data layer are:

This layer represent most data related artifacts, but not all. Blockchains are all about data and processing of data in terms of IO and deliverables.



Blockchain Architecture

The data layer

The data model can be very simple and contain just one asset, such as a cryptocurrency like Bitcoin, or a more complex model with multiple assets that can even have relationships between them.

An asset(s) can be created or referenced in a *transaction*, which in essence transfers the asset(s) between two parties who wish to exchange the data, for example, processing a payment between two parties, placing an order on an online store, registering an automobile, tracking diamonds around the world, or sharing your digital identity.

The chain structure is also related to transaction data. It describes the data structure in which individual transactions are combined into a block and how these blocks are chained to each other.

Blockchain Architecture

The network layer

The second layer up on the stack, just above the data layer, is the network layer. This layer deals with the propagation or broadcast of transactions and block data among available peers in the network, the reliability of the network, and local validation of data.

The network layer of a blockchain is similar to *BitTorrent*, and it is also managed by a peer-to-peer network, which is an architecture for distributing data in a network. The following diagram shows the common capabilities that are part of this layer:

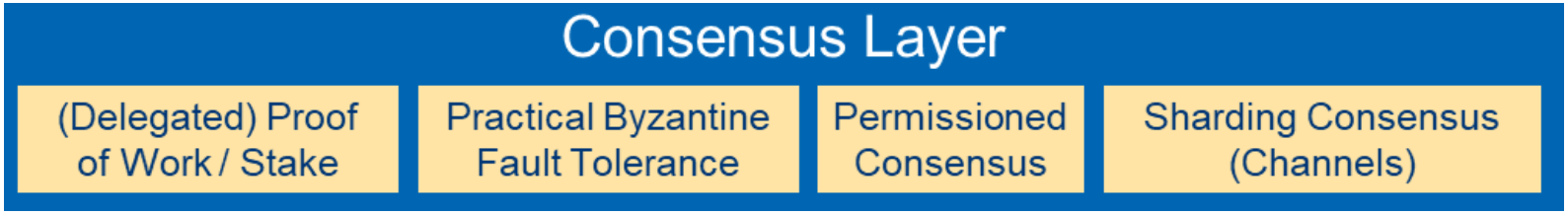
The capabilities of the network nodes varies in attributes based on public vs. private, permissioned or permissionless type.

Blockchain Architecture

The consensus layer

This layer deals with the enforcement of network rules that describe what nodes within the network should do to reach consensus about the broadcasted transactions. It also deals with the generation and verification of blocks.

The following diagram shows the common capabilities that are part of this layer:



This layer describes the rules for reaching consensus. The rules that need to be enforced depend on the consensus mechanism that is chosen when the network is initially set up.

Blockchain Architecture

Sophisticated consensus mechanisms

The **Proof of Work** (**PoW**) mechanism is used for consensus.

PoW used in the Bitcoin white paper as it allows for *trustless* and *distributed* consensus.

PoW requires participating nodes to perform an intensive form of calculations (mining)

The mining of transactions is necessary for two reasons:

- Verifying the legitimacy of transactions and record it permanently
- Creating new digital currency to reward first finished miner

Verified blocks of transactions are permanently added to the public blockchain ledger, and with every new block, the puzzle gets a bit more difficult.

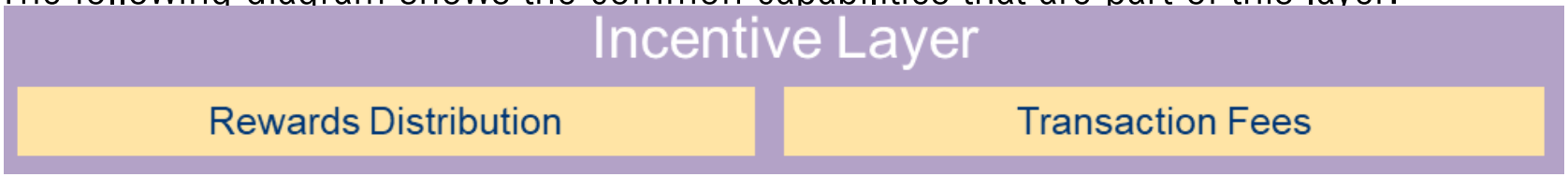
This requires miners to work more efficiently over time, this process consumes lots of power.

Blockchain Architecture

The incentive layer

This 4th layer deals with the distribution of rewards (for mining) that are earned by nodes in the network for the work they do to reach consensus. Whether this layer is implemented or not depends on the consensus mechanism in use.

The following diagram shows the common capabilities that are part of this layer:



It includes capabilities that describe what kinds of incentives are given by the network, when and how incentives can be earned by nodes, and the minimum amount of transaction fees (gas) needed to perform actions on the blockchain.

Blockchain Architecture

The application layer

The fifth (top) layer of the stack is called the application layer. This layer deals with providing the interfaces to access, program, and use the blockchain. The following diagram shows the common capabilities that are part of this layer:

The capabilities of this layer, including the programmable smart contracts and APIs.

The capabilities describe how the digital ledger is implemented and exposed to the world, how smart contracts can be built and run on the blockchain, and how third-party applications can interact with the digital ledger and smart contracts.

Blockchain Functions

Distributed ledgers

- A DL is a database that is synchronized and accessible across different sites and geographies by multiple P2P participants.
- The need for a central authority to keep a check against manipulation is eliminated by the use of a distributed ledger.
- A DL can be described as a ledger of any transactions or contracts maintained in decentralized form across different locations and nodes.
- Cyber attacks and financial fraud are reduced by the use of distributed ledgers.
- All the information on the ledger is securely and accurately stored using cryptography and can be accessed using keys and cryptographic signatures.
- Once the information is stored, it becomes an immutable database, which the rules of the network govern.

Blockchain Functions

Hyperledger Fabric

HYPERLEDGER FABRIC

Hyperledger Fabric is an open-source community enterprise-grade, distributed ledger platform that offers modularity and versatility for a broad set of industry use cases. The modular architecture for Hyperledger Fabric accommodates the diversity of enterprise use cases through plug and play components, such as consensus, privacy and membership services.

The key features of Hyperledger Fabric

- Permissioned architecture
- Highly modular
- Pluggable consensus
- Open smart contract
- Low latency of finality/confirmation
- Flexible approach to data privacy
- Multi-language smart contract support: *Solidity, Golang, Java, Javascript*
- Designed for continuous operations
- Governance and versioning of smart contracts
- Flexible endorsement model for achieving consensus across required organizations

Blockchain Functions

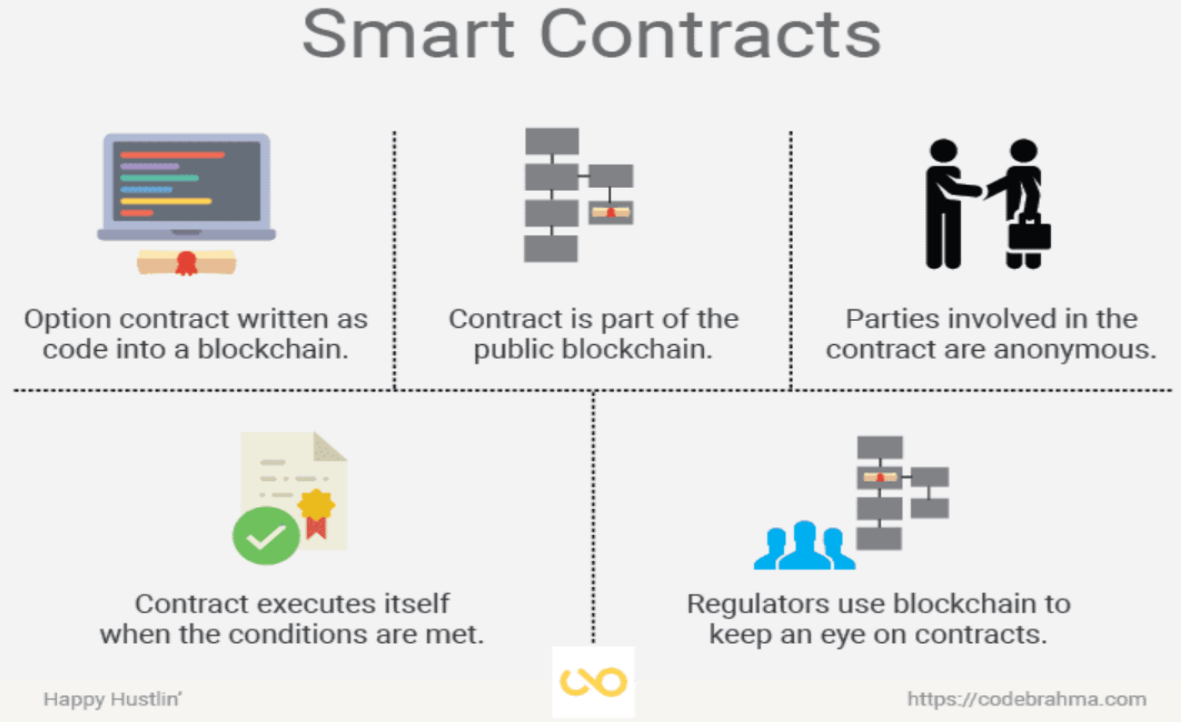
Smart contract

Smart contracts can act as a complement, or substitute, for legal agreements.

They are computer code that directly control some aspects of condition-based transactions.

A smart contract also capable of automatically facilitating, executing, and enforcing the negotiation or performance of an agreement.

Smart contracts are immutable and are enforced by the system itself.



Blockchain Functions

Decentralized applications

A capability that is still a very new concept is a decentralized application.

A **decentralized application** (**dApp**) is a blockchain-enabled website that runs independently on every node of the peer-to-peer network, rather than on a single serve.

They are comprised of both a frontend (web) application and a backend application, where the smart contract (backend application) allows it to connect to the blockchain.

For example, a decentralized application includes the data model it uses (participants, assets, and transactions), an authorization and permissions model, smart contracts (backend), and a frontend web application.

Industrial Case

Blockchain-Based Smart Contract E-Voting for National Elections*

Objective

A highly secured E-voting is one of the valid use cases of blockchain technology. This research involves elicitation of the e-voting requirements, formulation of a blockchain e-voting architecture, an architecture-based evaluation, the analysis of the results, and a report of the findings.

* Architecture-Centric Evaluation of Blockchain-Based Smart Contract E-Voting for National Elections
Olawande Daramola, and Darren Thebus, Informatics, May 2020
Department of Information Technology, Cape Peninsula University of Technology, Cape Town

Smart Contract E-Voting Requirements

The following are the system key stakeholders requirements:

1- Trust: All stakeholders must have confidence and trust the SCE outcome

2- Transparency: System supports the casting of votes and tally of votes by all stakeholders, as well as allow them to verify this easily.

3- Verifiability: The system must enable voters to check that their votes were cast and recorded as valid votes.

4- Auditability: Support any process that may necessitate the rechecking

and recounting votes in the event of electoral disputes.

5- Availability: Backup system ensuring almost zero-down time.

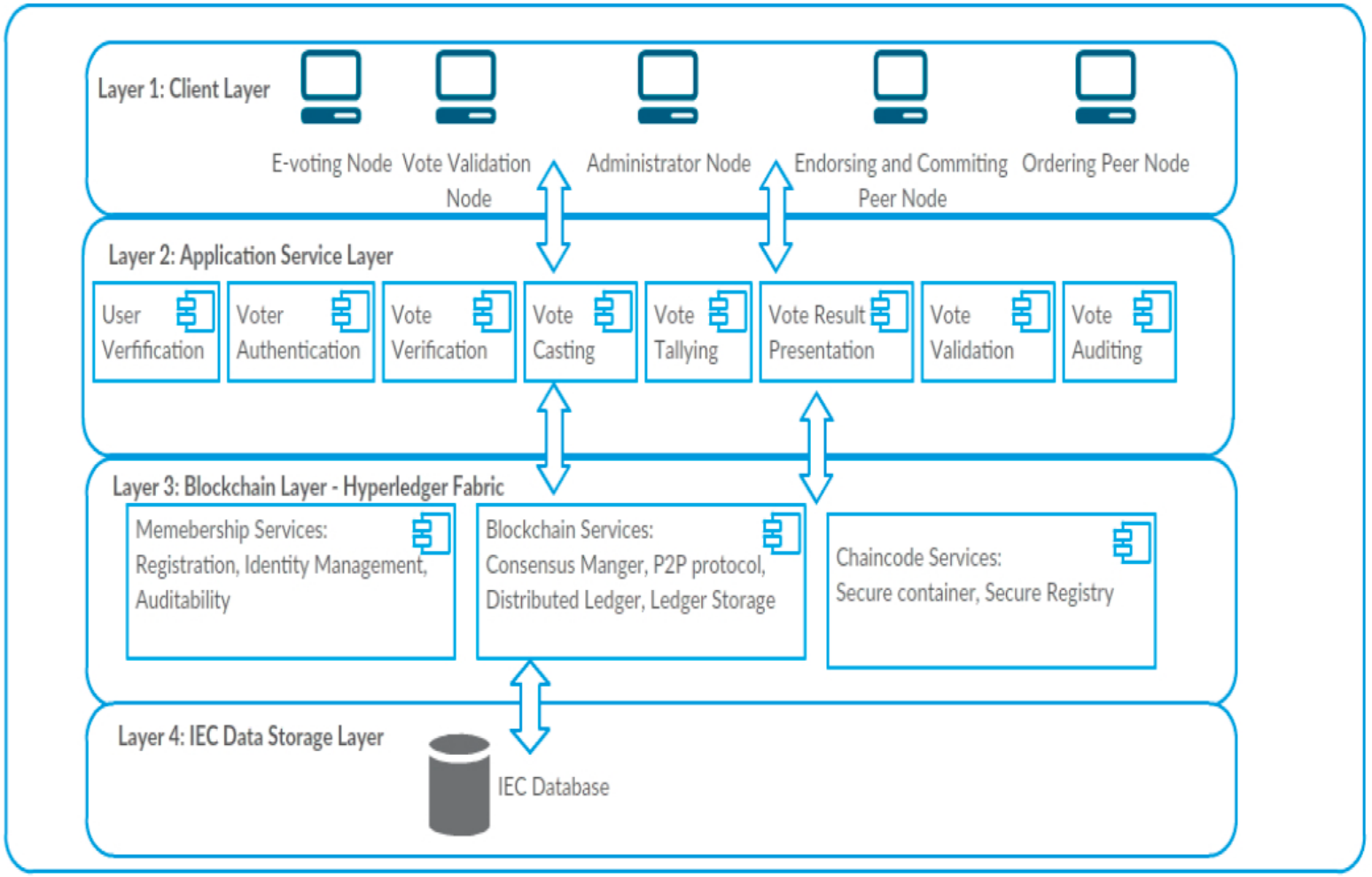
6- Performance: Ensure that all operations are handled speedily and efficiently.

the identity of voters, and the choices made during voting.

7- Socio-political factors: The e-voting system should not be vulnerable to socio-political manipulations that can compromise the integrity of the voting process.

BANES Architecture

Based on the identified requirements, a Blockchain-based Architecture for National E-voting system (BANES) a Layered Architecture was proposed, as shown below in Fig.1



Blockchain style within Layered Architecture Style

Figure 1. A schematic view of the blockchain-based architecture for national e-voting system (BANES).

BANES Architecture

Client Layer

This layer contains the various electronic devices and systems with which users interact with the blockchain e-voting system. These devices are the peer nodes of the e-voting blockchain that interact via smart contracts, referred to as "chaincode" in the Hyperledger Fabric.

(i) E-Voting nodes: enable voters authentication and casting of votes, and to ensure that all blockchain transactions are recorded.

(ii) Administrator nodes: used to configure blockchain network channels, assign roles to the nodes of the blockchain, and grant permissions.

(iii) Public nodes: enable public view-only to transactions of the e-voting blockchain.

(iv) Vote validation: responsible for vote validation. They are also used to ensure the authenticity of transactions that are included in a block.

(v) Committing nodes: These are the nodes that validate and commit new blocks to the blockchain.

BANES Architecture

Application Service Layer

Consists of a set of services that are available in the e-voting system. The level of access control and the defined permissions level determines the type of services that a node can access in the blockchain.

Blockchain Layer

It is composed of the Hyperledger Fabric V2.0, which is a modular blockchain architecture framework that facilitates blockchain information system solutions.

It supports the creation of permissioned blockchain networks that have built-in properties such as security, and privacy protection.

The Hyperledger Fabric has “ordering nodes” which ensures consistency of the blockchain by ensuring that only ordered blocks of an endorsed transaction are made available to the committing peer nodes before they are added to the blockchain.

As we stated before,

BANES Architecture

As stated before, the use of Hyperledger fabric is motivated by:

- Permissioned architecture
- Highly modular
- Pluggable consensus
- Open smart contract
- Low latency of finality/confirmation
- Flexible approach to data privacy

IEC* Data Storage Layer

It contains the relevant databases that store information on the profile of registered voters. This database is used as the basis to authenticate and authorize voters to vote.

* ISO/IEC 27040 is to provide security guidance for storage systems and ecosystems as well as for protection of data in these systems.