IN4315: Software Architecture

Architecting for Scalability

4

Prof. Diomidis Spinellis

http://www.spinellis.gr/
D.Spinellis@tudelft.nl

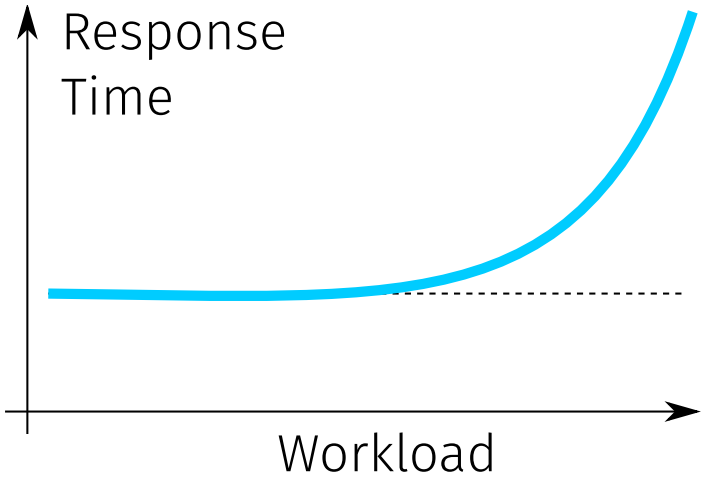
Based on material by Prof. Cesare Pautasso
http://www.pautasso.info/
cesare.pautasso@usi.ch

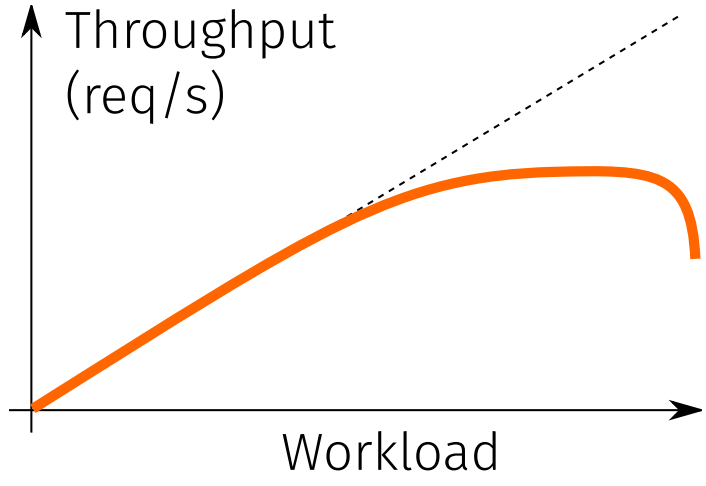
Contents

- Scalability: Workloads and Resources
- Scale up or Scale out?
- Scaling Dimensions: Number of Clients (Workload), Input Size, State Size, Number of Dependencies
- Location Transparency: Directory and Dependency Injection
- Scalability Patterns: Scatter/Gather, Master/Worker, Load Balancing, and Sharding


Will it scale?
powered by
ASQ

Scalability and Workload



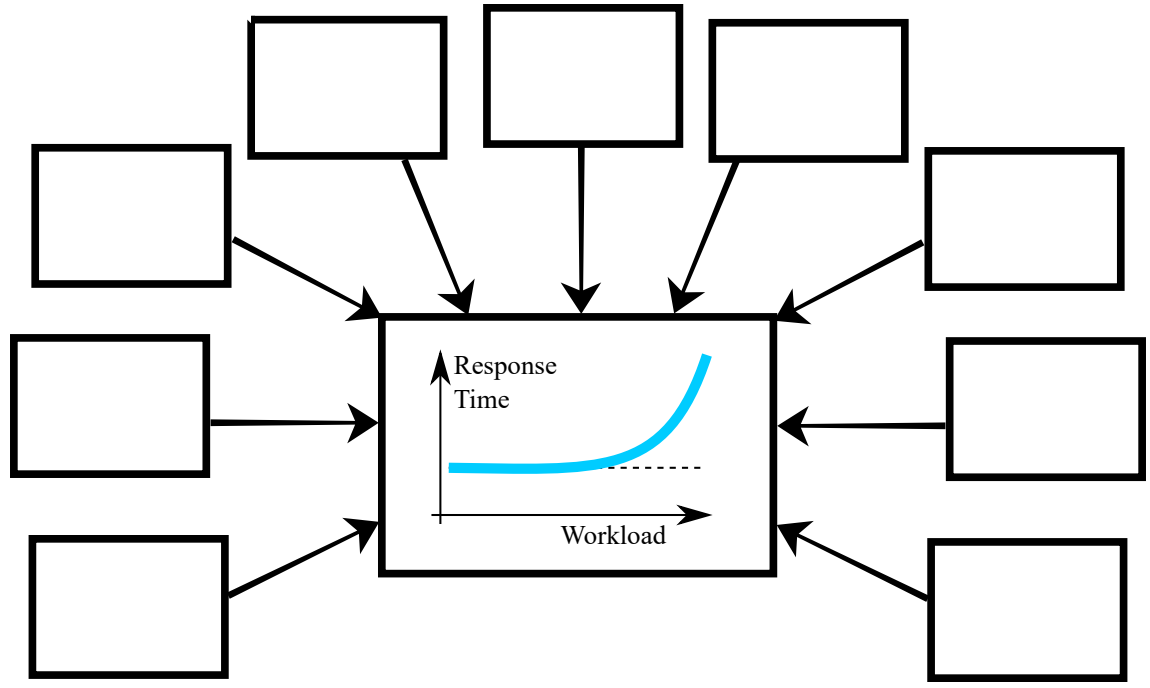


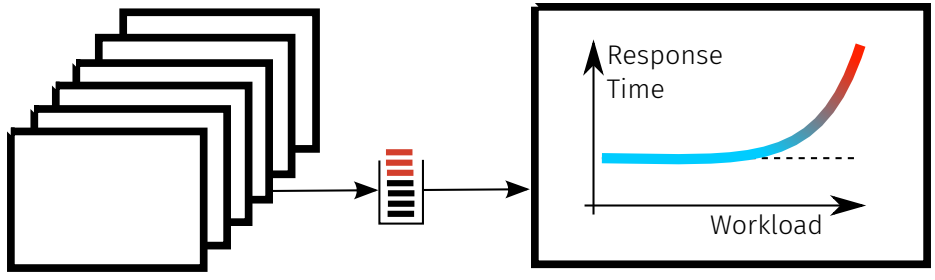
Workload = traffic, number of clients or their number of concurrent requests

Ideal system: response time not affected by the workload; throughput grows proportionally to the workload

Real systems will show this behavior up to their capacity limit

Scalability and Workload: Centralized



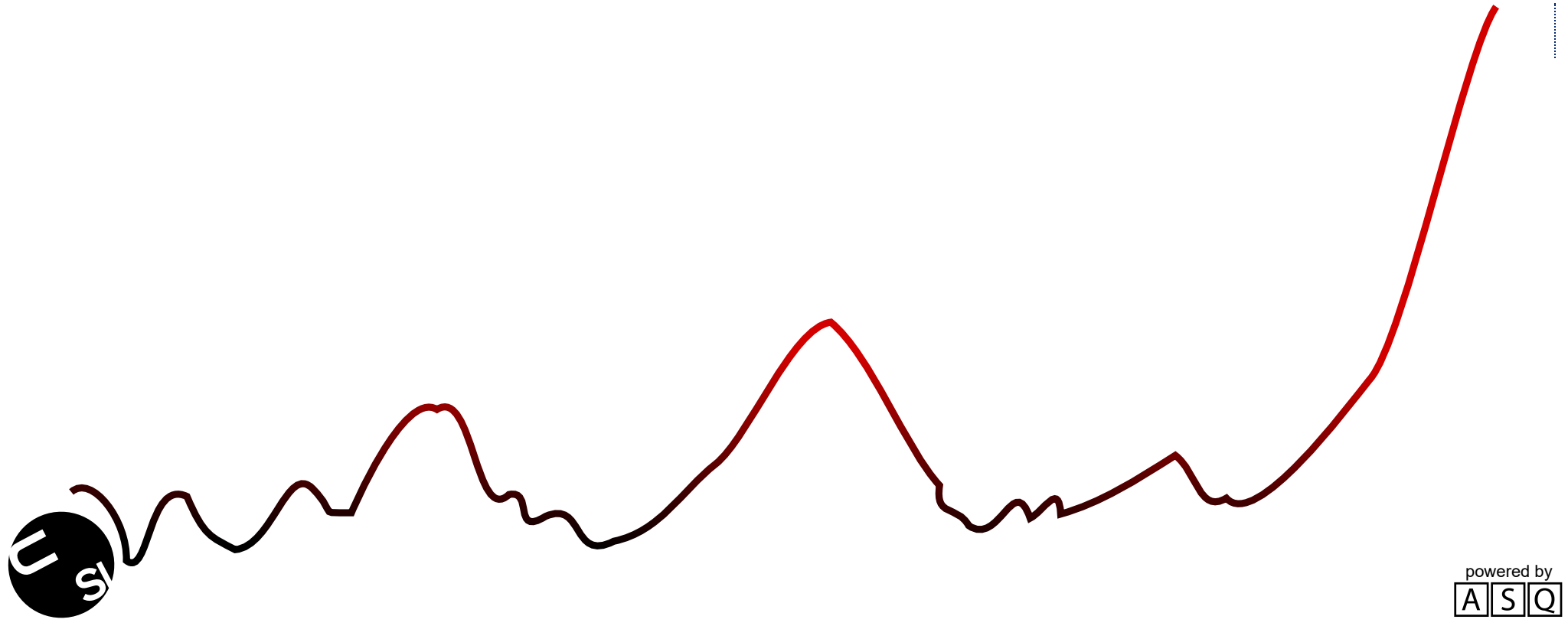


Many 1

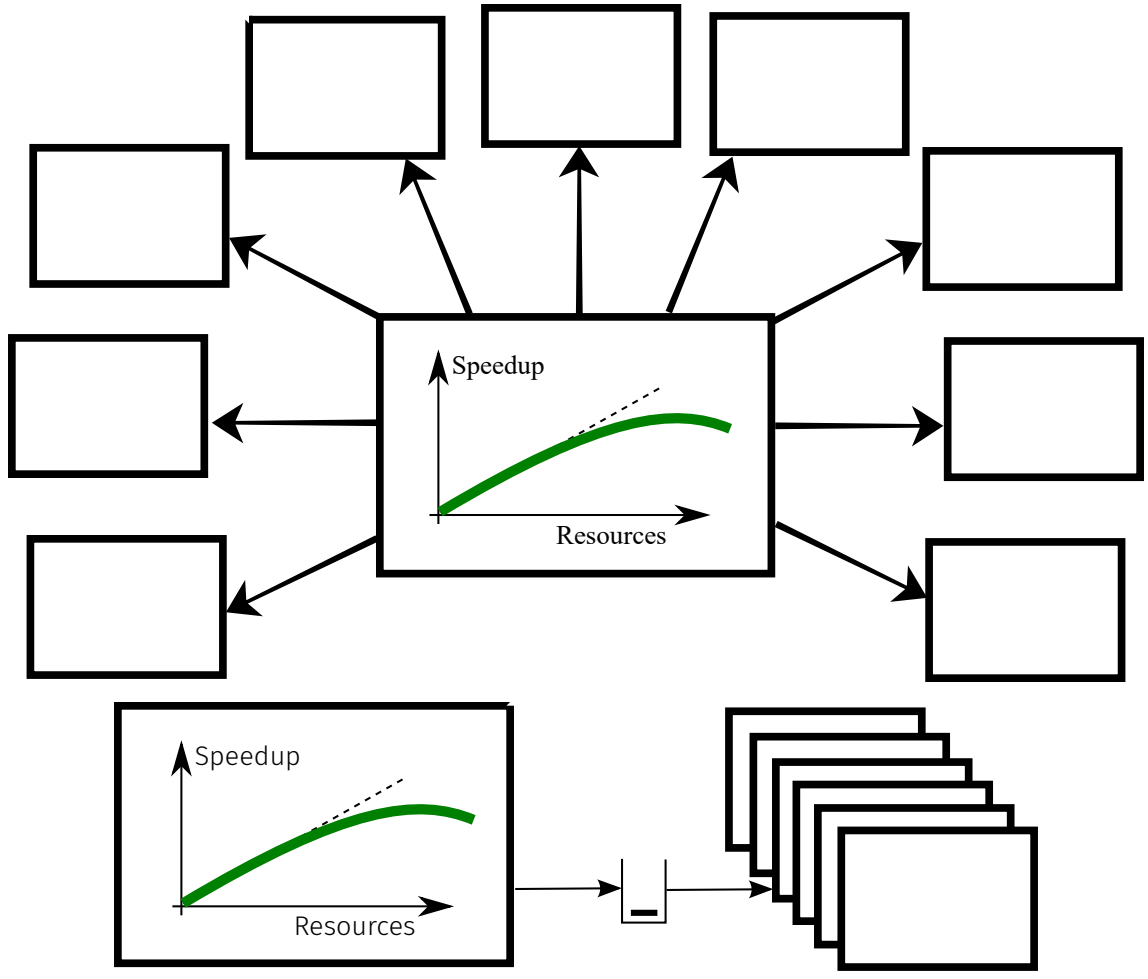

Done
How to scale?
powered by
ASQ

How to scale?

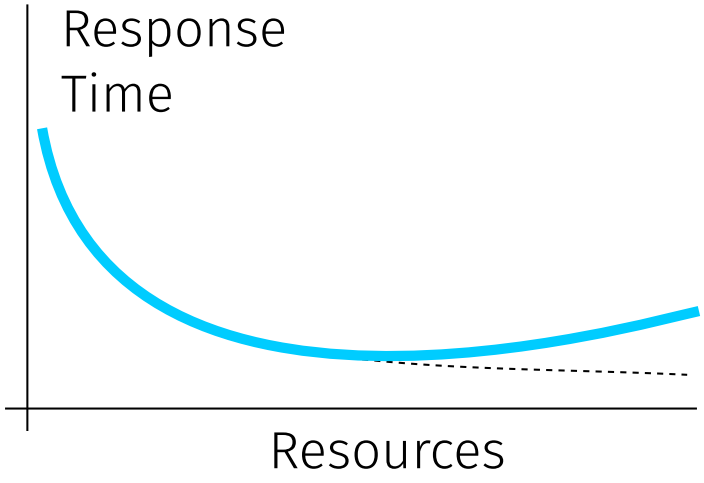
1. Work faster: better algorithms, €€€, Moore's Law
2. Work less: approximations, caching, client offloading
3. Work later: queuing
4. Get help: ...

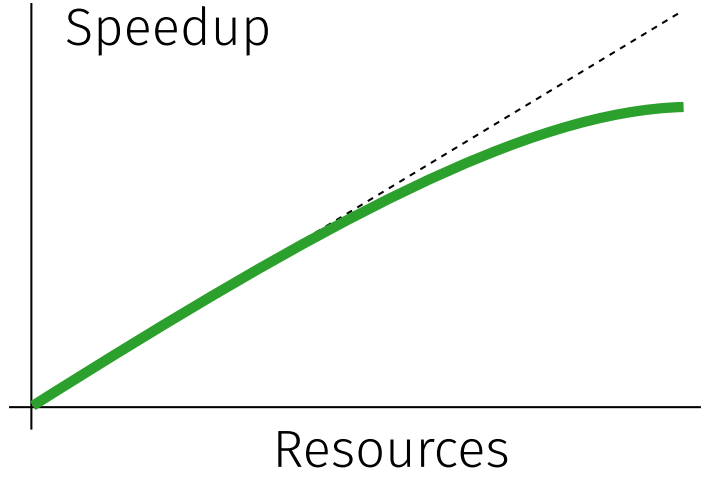


Scalability and Resources: Decentralized



Scalability and Resources





For the same workload, will performance improve by adding more resources?

Ideal system: linear speedup with infinite resources

Real systems will only benefit up to a limited amount of resources

Done

Centralized or Decentralized?

Centralized	Decentralized

Client/Server

Peer to Peer

Single Point of Failure

Hot Spot

Consistent

Bottleneck

Churn

Partial Failure

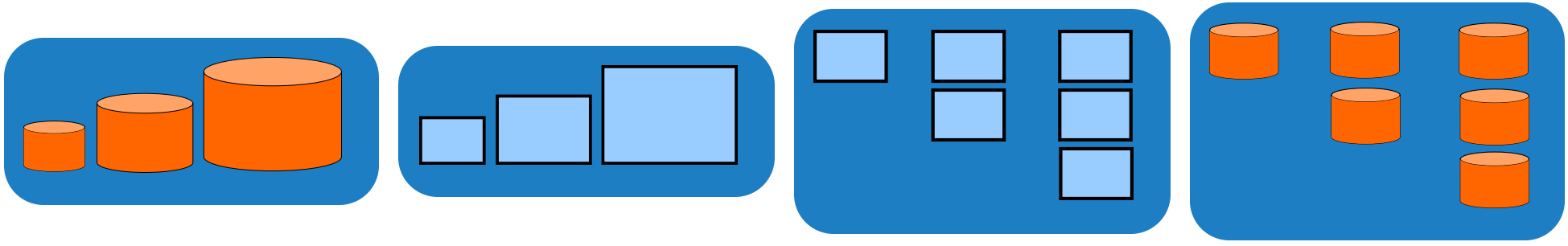
Scalability at Scale

Done

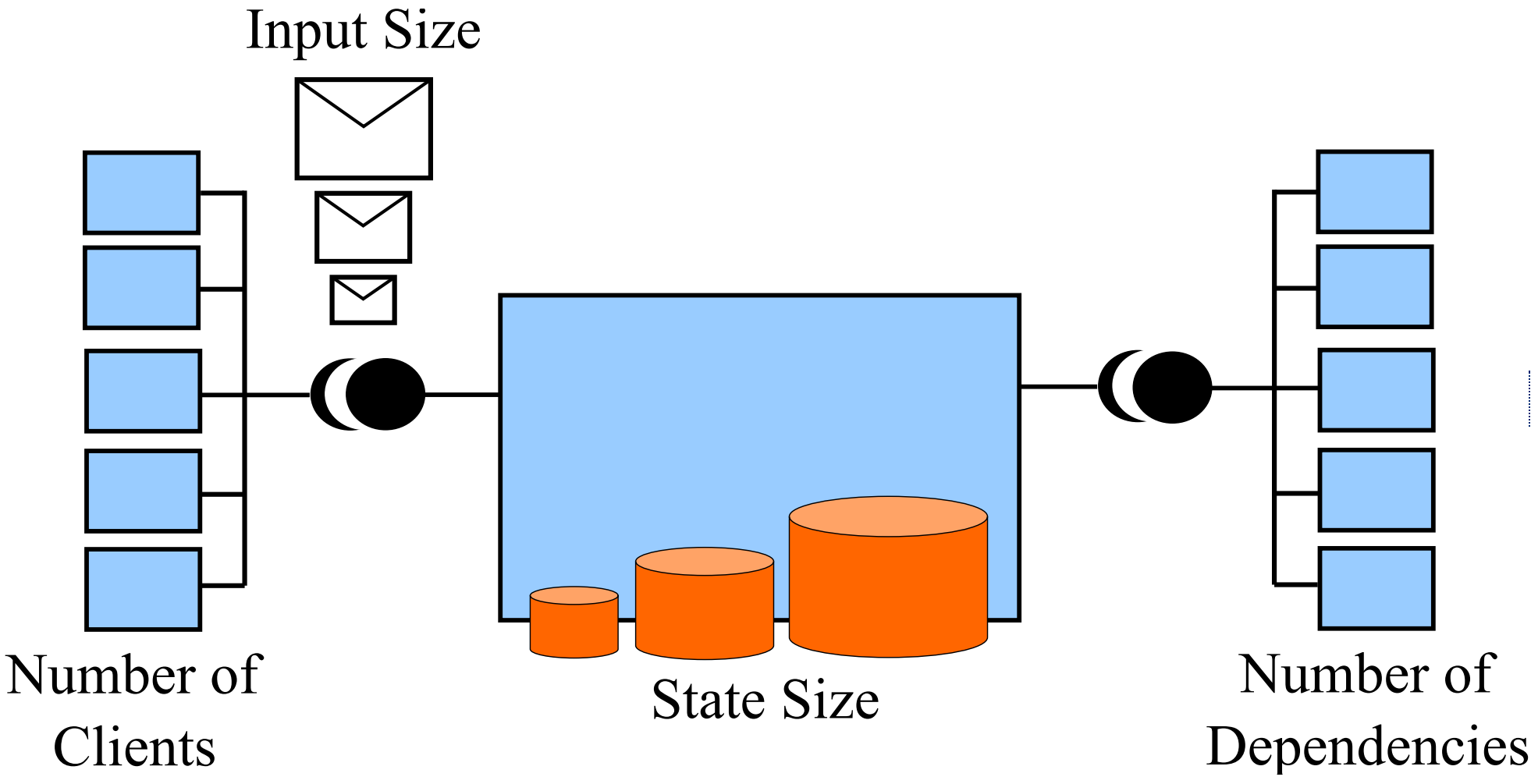
Scale Up or Scale Out?

Scale Up

Scale Out



Scaling Dimensions



Scalability Patterns

# Component Dependencies	Directory Dependency Injection
# Clients (Workload)	Load Balancing
Input Size	Master/Worker Scatter/Gather
State Size	Sharding

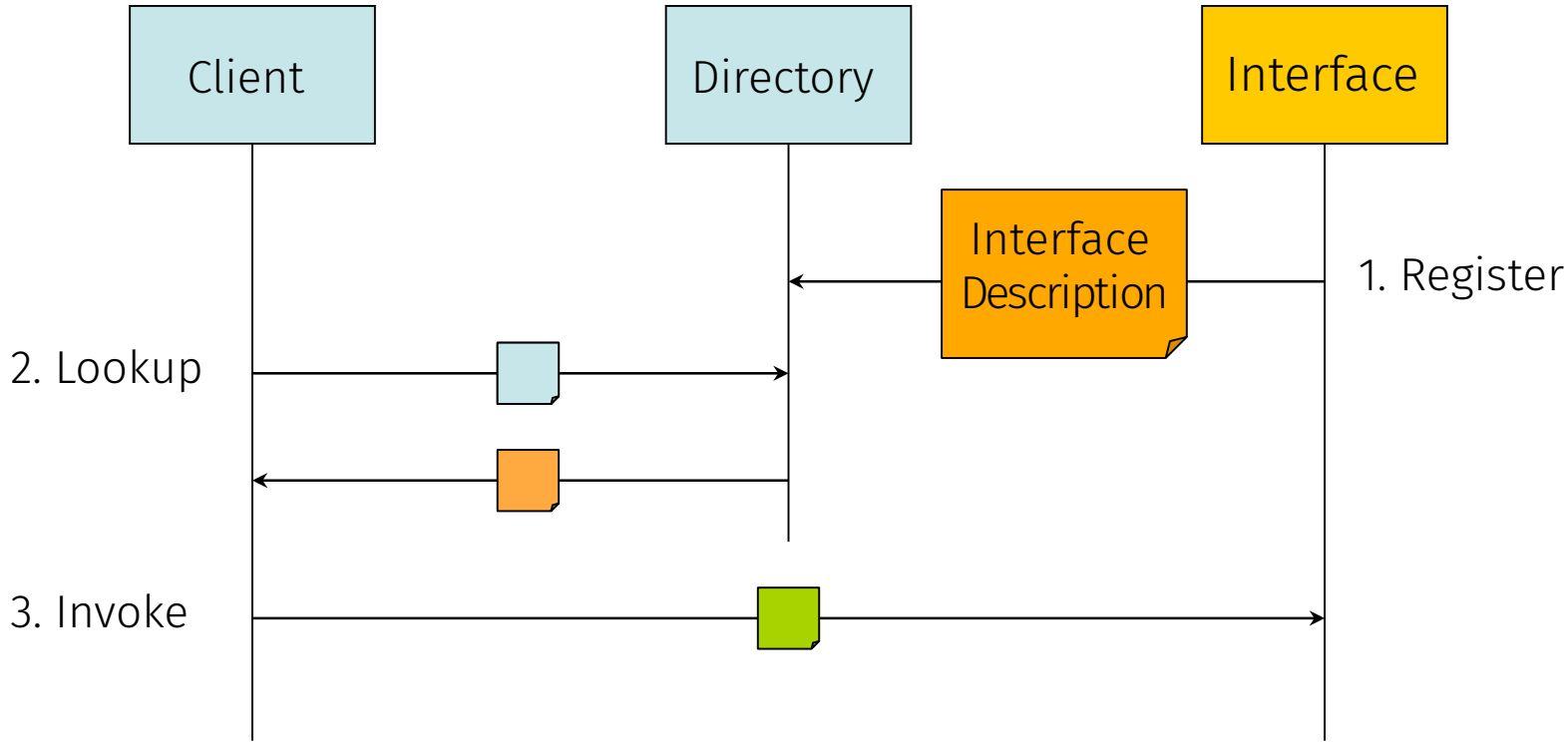
Directory

How to facilitate location transparency?

use a directory to find interface endpoints based on abstract descriptions

Clients avoid hard-coding knowledge about required interfaces as they **lookup their dependencies through a directory** which knows how and where to find them

Directory



Clients use the Directory to lookup published interfaces descriptions that will enable them to perform the actual invocation of the component they depend on

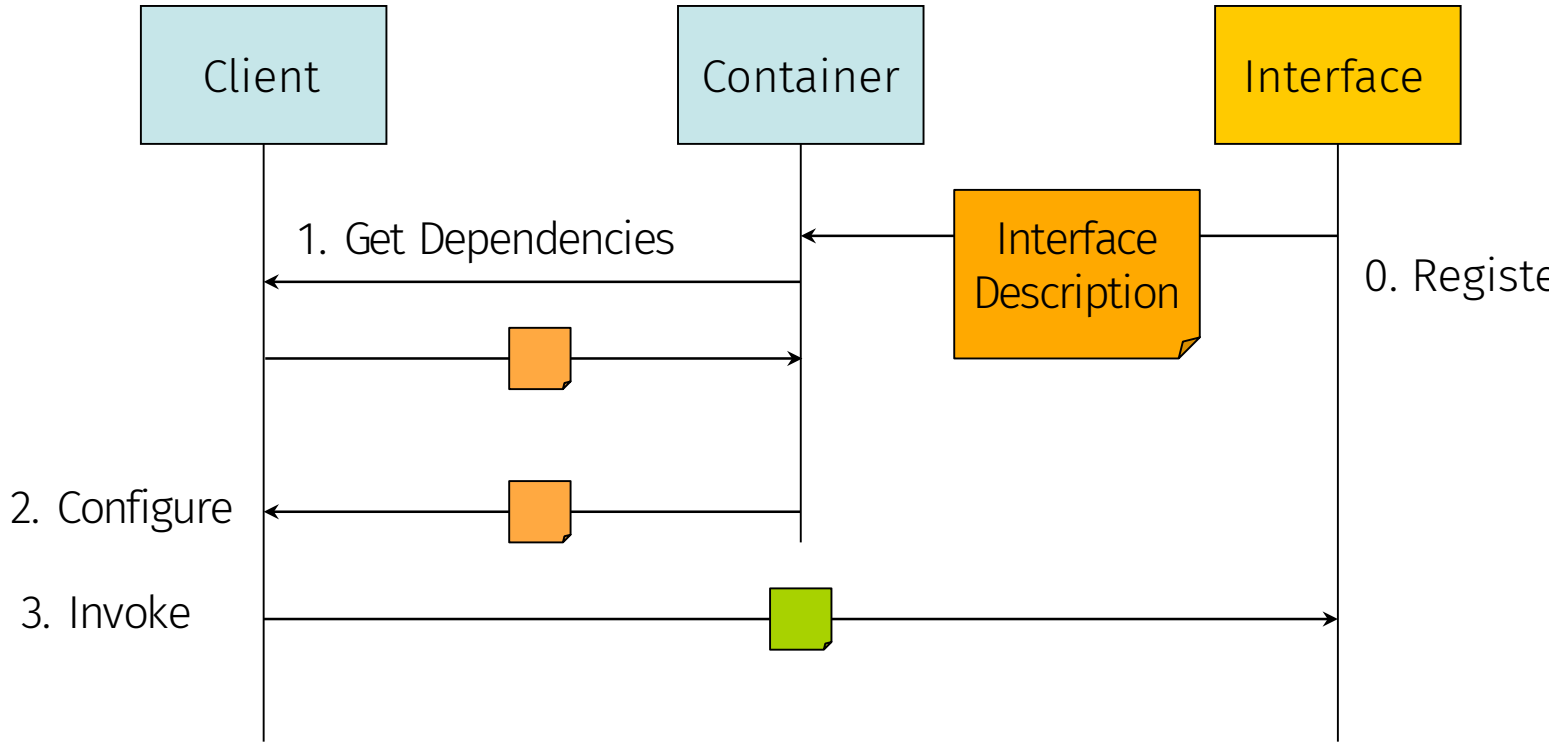
Dependency Injection

How to facilitate location transparency?

use a container which updates components with bindings to their dependencies

Clients avoid hard-coding knowledge about required interfaces as they expose a mechanism (setter or constructor) so that they can be configured with the necessary bindings

Dependency Injection



As components are deployed in the container they are updated with bindings to the interfaces they require

Dependency Injection

- Used to design architectures that follow the inversion of control principle:
  - “don't call us, we'll call you”, Hollywood Principle
- Components are passively configured (as opposed to actively looking up interfaces) to satisfy their dependencies:
  - Components should depend on required interfaces so that they are decoupled from the actual component implementations (which may be changed anytime)

Dependency Injection

- Flexibility:
  - Systems are a loosely coupled collection of components that are externally connected and configured
  - Component bindings can be reconfigured at any time (multiple times)
- Testability:
  - Easy to switch components with mockups

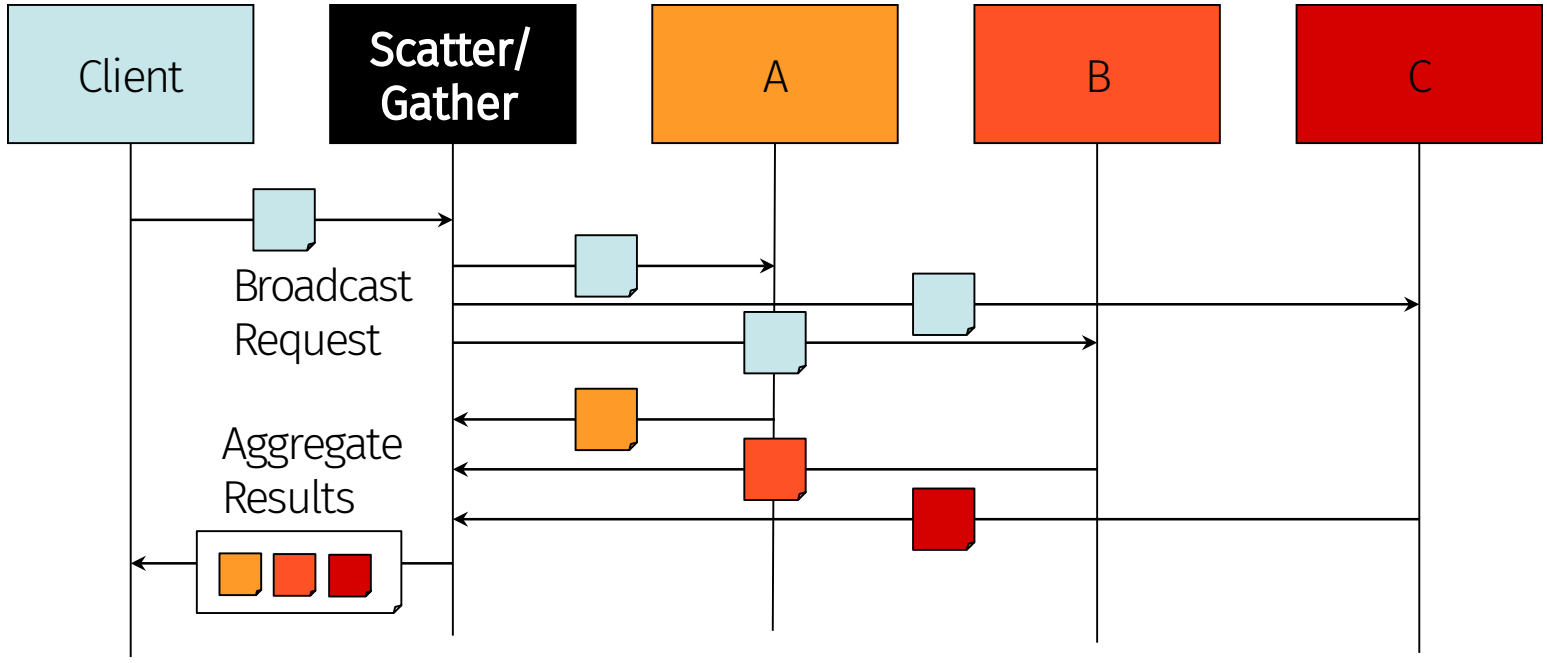
Scatter/Gather

How to compose many equivalent interfaces?

Broadcast the same request and aggregate the replies

Send the same request message to all recipients, wait for all (or some) answers and aggregate them into a single reply message

Scatter/Gather



Example:

- Contact N airlines simultaneously for price quotes
- Buy ticket from either airline if price <= 200 CHF
- Buy the cheapest ticket if price > 200 CHF
- Make the decision within 2 minutes

Scatter/Gather

Which components should be involved?

- The recipients are kept hidden from the client
  They can be dynamically discovered using subscriptions or directory registrations
- The recipients are known a priori by the client
  The request includes a distribution list with the targeted client addresses

Scatter/Gather

How to aggregate the responses?

- Send all (packaged into one message)
- Send one, computed using some aggregation function (e.g., average)
- Send the best one, picked with some comparison function
- Send the majority version (in case of discrepancy)

Scatter/Gather

Synchronization strategies

- When to send the aggregated response?
  - Wait for all messages
  - Wait for some messages within a certain time window
  - Wait for the first N out of M messages (N < M)
  - Return fastest acceptable reply
- Warning: the response-time of each component may vary and the response-time of the scatter/gather is the slowest of all component responses

Master/Worker

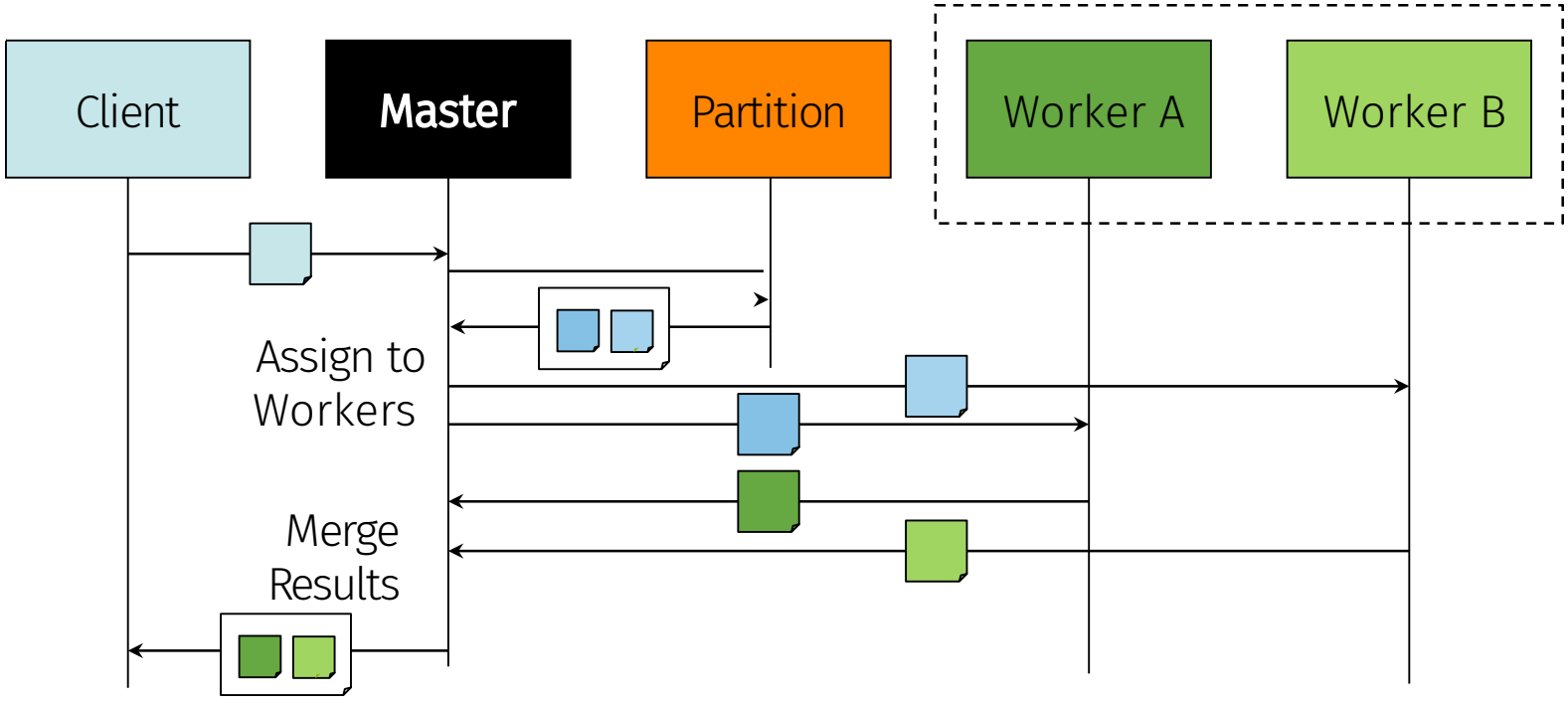
How speed up processing large amounts of input data?

split a large job into smaller independent partitions which can be processed in parallel

The master divides the work among a pool of workers and gathers the results once they arrive

Synonyms: Master/Slave, Divide-and-Conquer

Master/Worker



Example:

- Matrix Multiplication (compute each row independently)
- Movie Rendering (compute each picture frame independently)
- Seti@home (volunteer computing)

Master/Worker

Master Responsibilities

- Transparency: Clients should not know that the master delegates its task to a set of workers
- Partitioning strategies: Uniform, Adaptive (based on available worker resources), Static/Dynamic
- Fault Tolerance: if a worker fails, resend its partition to another one
- Computational Accuracy: scatter the same partition to multiple workers and compare their results to detect inaccuracies (assuming workers are deterministic)
- Master is application independent

Master/Worker

Worker Responsibilities

- Each worker runs its own parallel thread of control and may be **distributed** across the network
- Worker **churn**: they may join and leave the system at any time (may even fail)
- Workers do not usually exchange any information among themselves
- Workers should be independent from the algorithm used to partition the work
- Workers are application domain-specific

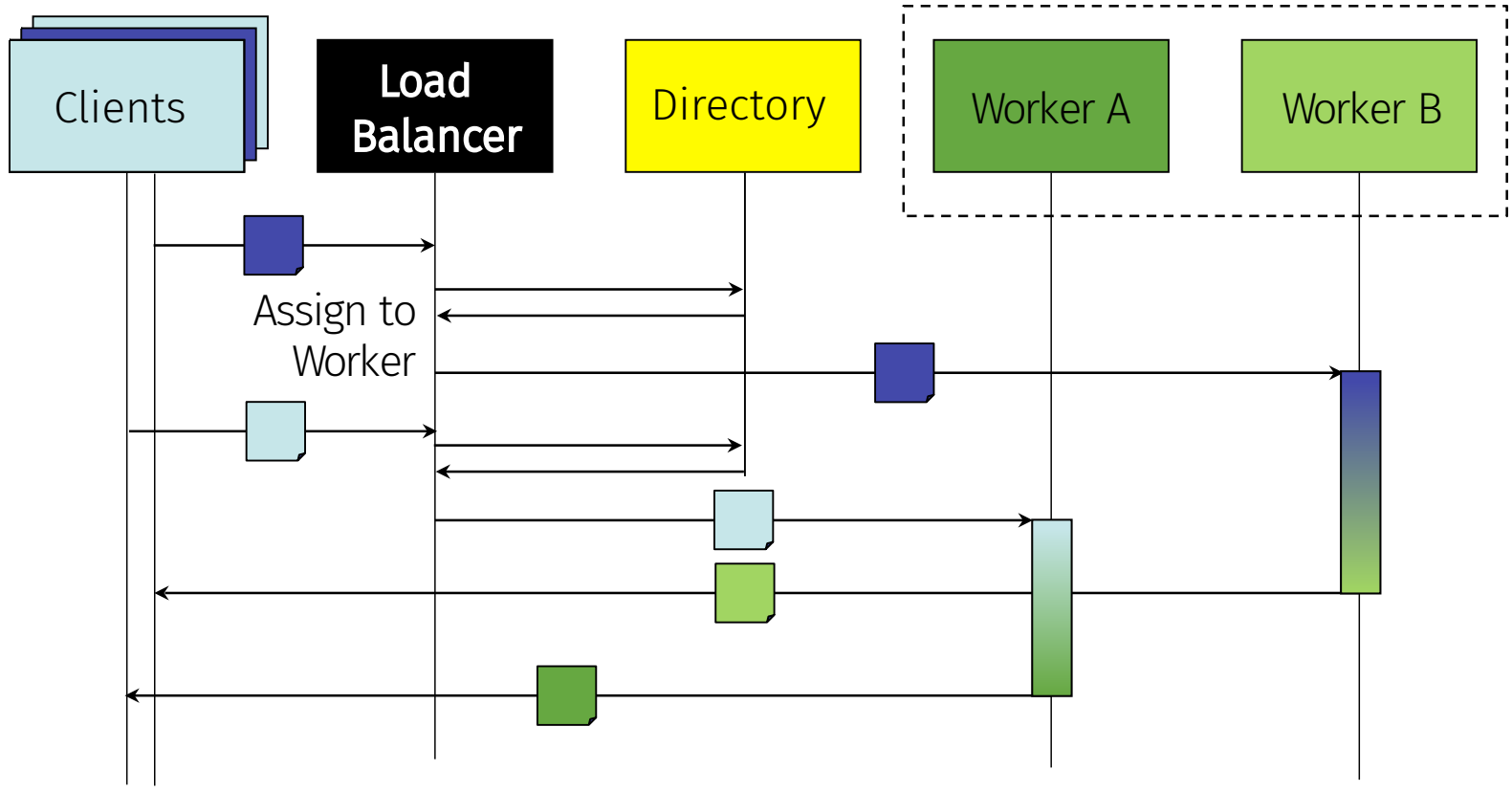
Load Balancing

How to speed up processing multiple requests of many clients?

deploy many replicated instances of stateless components on multiple machines

The Load Balancer routes requests among a pool of workers, which answer directly to the clients

Load Balancing



Load Balancing

Strategies

- Round Robin
- Random
- Random Robin
- First Available
- Nearest

Location

- Server (transparent from client)
- Client (aware of choice between alternative workers)
- Directory (e.g., DNS)

Layer

- Hardware (network device)
- Software (OS, or user-level)

Load Balancing

Variants

- **Stateless**: every request from any client goes to any worker (which must be stateless)
- **Session-based**: requests from the same client always go to the same worker (which could be stateful)
- **Elastic**: the pool of workers is dynamically resized based on the amount of traffic

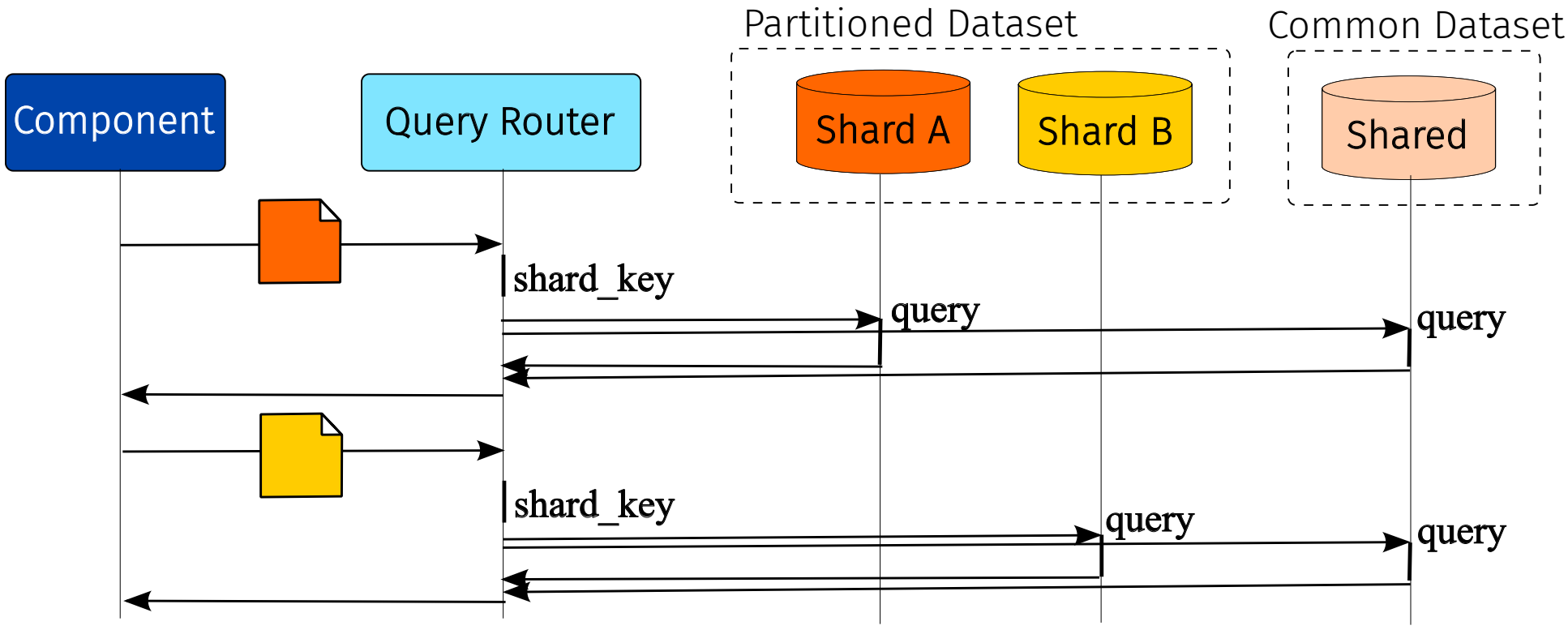
Sharding

How to scale beyond the capacity of a single database?

partition the data across multiple databases

Route the queries to the corresponding data partition ensuring each partition remains independent and balanced

Sharding



Queries are directed to the corresponding shard (partition)

Queries should not involve more than one independent shard, even if they may use some shared non-sharded dataset that may need to be replicated on each shard

Sharding

- Two goals:
  - scale capacity (storage size, bandwidth, concurrent transactions)
  - balance query workload (avoid hotspots)
- What to shard?
  - Large data collections (they do not fit)
  - Hot spots (increase throughput of queries targeting a subset of the data)

Sharding

Computing the Shard Key

- Key Ranges
- Geo-spatial (country shards)
- Time Range (year, month, day)
- Hash Partitioning
- Modulo (Number of Shards)

Usually assumes a fixed and static number of shards

Sharding

Looking up the Shard Key

- Business domain-driven (customer/tenant id)
- Helps to keep data balanced
- Requires to maintan a master index (i.e., a directory for shards, a ZooKeeper)

Sharding

- Changing the number of shards or the sharding strategy may require an expensive and complex repartitioning operation
- Transactions should only involve one shard (some shared data may need to be replicated on each shard)
- Sharding was originally implemented outside the data layer, sometime as part of the application logic. Some databases are starting to offer native sharding support

Sharding

Different systems use different terms to name data partitioning for scalability

Shard	MongoDB, Elasticsearch, SolrCloud
Region	HBase
Tablet	Bigtable
vnode	Cassandra, Riak
vbucket	Couchbase

References

- Martin L. Abbott, Michael T. Fisher, The Art of Scalability, Pearson, 2015
- Gregor Hohpe and Bobby Woolf, **Enterprise Integration Patterns**, Addison-Wesley, October 2003, ISBN 0321200683
- Martin Kleppmann, Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable and Maintainable Systems, O' Reilly, 2017, ISBN 978-1-449-37332-0