

## Advanced Distributed Systems

---

Karl M. Göschka  
Karl.Goeschka@tuwien.ac.at

[http://www.infosys.tuwien.ac.at/teaching/courses/  
AdvancedDistributedSystems/](http://www.infosys.tuwien.ac.at/teaching/courses/AdvancedDistributedSystems/)

## Replication

---

- Basic concepts and consistency models
- Design considerations
- Replication protocols (DS)
- Replication in DB, SOA, P2P
- Replication and middleware

## What is Replication?

---

Replication is the process of maintaining several copies of an entity (object, data item, process, file, ...) at different nodes.

## Reasons for Replication

---

- Fault tolerance (redundancy)
  - switch-over in case of failures
  - protection against corrupted data (→ voting)
- Performance (and scalability)
  - scale in numbers (cluster)
  - scale in geographic/topological complexity (place copies of data and processes in proximity)
- BUT: price for replication: keeping replicas consistent in face of updates is costly  
→ trade-off

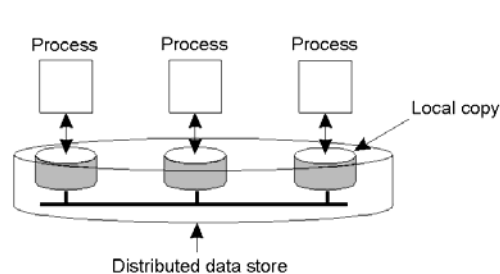
## Replication as scaling technique (1)

- Scalability issues: performance problems
  - access time (process close to data)
  - network bandwidth
  - server utilization
- Copy up-to-date  $\leftrightarrow$  network bandwidth
  - access-to-update ratio (read to write ratio)
- Consistency itself may impose scalability problems (e.g. **atomic** update; global order or coordinator):  
Scalability problems  $\rightarrow$  replication  $\rightarrow$  consistency  $\rightarrow$  scalability problems:  
The cure may be worse than the disease!

## Replication as scaling technique (2)

- Solution: **relax** the consistency requirements!
  - depends on access and update patterns
  - depends on purpose of the data
- **Consistency model**: *contract* between processes and the distributed data store
  - Each model effectively restricts the values that a read operation on a data item can return with respect to different (previous) write operations
  - **Data centric** vs. **Client centric (mobility support)**
  - Consistency models define what behaviour exactly is acceptable in the presence of **conflicts**

## Data-Centric Consistency Models



The general organization of a logical data store, **physically distributed and replicated** across **multiple machines**.

## Data-Centric Consistency Model

- Normally one would like: “*read* returns the result of latest *write*”
- However: No global clock! What is **last** write?
- We **relax** this model by considering **time intervals** and define precisely what is **acceptable behaviour** for **conflicting** operations
- **Conflict**: Two operations in the same interval on the same data item and at least one is a write.

## Sequential Consistency

Sequential Consistency:

The **result** of any execution is the **same as if**

1. the operations by all processes on the data store were executed in **some sequential order** and
2. the operations of each individual process appear in this sequence in the **order specified by its program**

Possible valid sequence:

$W_2(x)b, R_3(x)b, R_4(x)b, W_1(x)a, R_3(x)a, R_4(x)a$

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)b	R(x)a

(a)

a) A sequentially consistent data store.

b) A data store that is not sequentially consistent.

No single valid sequence can be constructed

P1:	W(x)a		
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

## Causal Consistency (2)

P1:	W(x)a		
P2:		R(x)a	→ W(x)b
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(a)

a) The writes are causally related, thus this is a **violation** of a causally-consistent store.

b) The writes are not causally related, thus this is a **correct** sequence.

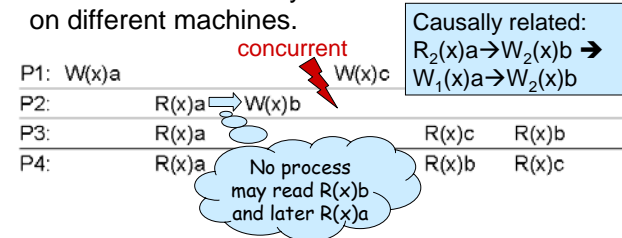
P1:	W(x)a		
P2:		W(x)b	
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

## Causal Consistency (1)

- Necessary condition:

- Writes that are **potentially causally related** must be seen by all processes in the same order.
- **Concurrent writes** may be seen in a different order on different machines.



- This sequence is allowed with a causally-consistent store, but not with a sequentially consistent store.

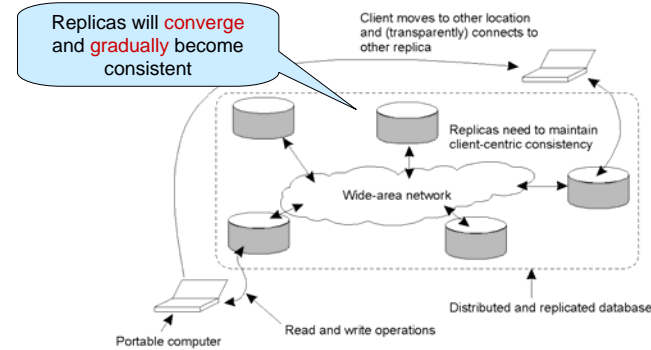
## Client-Centric Consistency Models

- So far: Concurrent processes require simultaneous updates of shared data → consistency and isolation have to be maintained → synchronization required
- But: Some distributed data stores are characterized by the **lack of simultaneous updates** (or they can easily be resolved or the resulting inconsistencies can cheaply be hidden or are simply acceptable); most operations involve reading only: e.g. DNS, WWW, most information systems, ...
- → Guarantees for a **single (mobile) client**, but not for concurrent access!

## Eventual Consistency

- Update is performed at one replica
- Propagation to other replicas is performed in a lazy fashion
- Eventually, all replicas will be updated
- I.e., replicas gradually become consistent if no updates take place for a long time

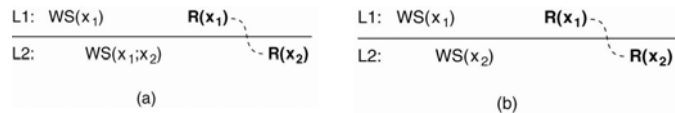
## Eventual Consistency



- The principle of a mobile user accessing *different* replicas of a distributed database.
- Data items have associated **owner**, no write-write conflicts
- R/W "locally", eventual **propagation** → 4 consistency models

## Monotonic Reads

If a process has seen a value of a data item  $x$  at time  $t$ , it will **never see an older version** of  $x$  at a later time.

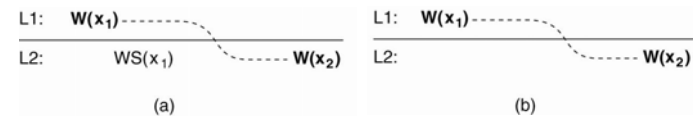


The read operations performed by a single process  $P$  at two different local copies of the same data store.

- a) A monotonic-read consistent data store
- b) A data store that does not provide monotonic reads.

## Monotonic Writes

If an update on a copy of a data item  $x$  is performed by a process, all **preceding updates** by the same process on  $x$  will be performed **first**.

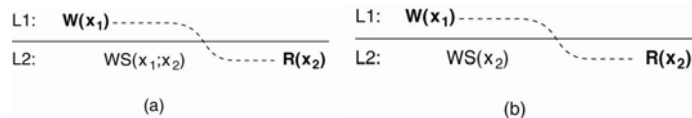


The write operations performed by a single process  $P$  at two different local copies of the same data store

- a) A monotonic-write consistent data store.
- b) A data store that does not provide monotonic-write consistency.

## Read Your Writes

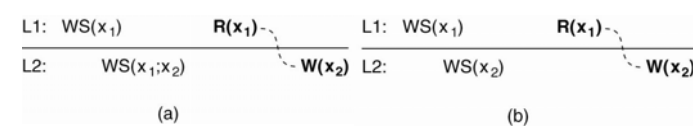
A **write** operation is always **completed** before a **successive read operation** by the same process, no matter where the read operation takes place.



- a) A data store that provides read-your-writes consistency.
- b) A data store that does not.

## Writes Follow Reads

Any successive **write operation** by a process on a data item  $x$  will be performed **on a copy of  $x$  that is up to date** with the value **most recently read** by that process.



- a) A writes-follow-reads consistent data store
- b) A data store that does not provide writes-follow-reads consistency

## Consistency of Replicas

- When an object is replicated, the replicas must be kept consistent according to some model
- If there are no updates to the object, there is no consistency problem
- If the “access to update” ratio is high, replication should pay off
- If the “update to access” ratio is high, many updates may be never accessed
- Ideally, we should only update the replicas that are going to be accessed
- As a general rule, we try to keep a replica “close” to its clients

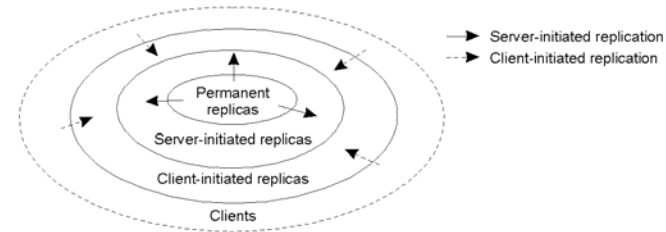
## Replication

- Basic concepts and consistency models
- Design considerations
- Replication protocols (DS)
- Replication in DB, SOA, P2P
- Replication and middleware

## Basic design considerations

- Replica server placement
  - often a management or commercial issue
- Replica (content) placement
- Update propagation
  - state vs. operation
  - pull vs. push vs. lease
  - blocking vs. non-blocking

## Replica Placement (1)



The logical organization of different kinds of copies of a data store into three concentric rings.

## Replica Placement (2)

- Major design issue!
- Permanent replicas
  - initial set (small)
  - LAN; e.g. Web server **cluster** or database cluster
  - geographically; e.g. Web **mirror** or **federated database**
- Server-initiated replicas
  - performance, e.g. **push cache** or **Web hosting service**
  - reduce **server load** and replicate to servers placed in the **proximity** of requesting clients
- Client-initiated replicas (often cache)

## Client-Initiated Replicas

- Clients can also create their own replicas (**cache**) → temporary copy
- Client cache improves **access times**
- **Maintenance** is up to the client (e.g. by polling)
- Data are kept for a **limited** amount of **time**
  - make room
  - avoid extremely stale copies
- Clients may **share** a cache (based on the assumption that clients access similar data)
- Placement: client, LAN, dedicated, ...

## Update propagation

- When there is an update, what is propagated to the replicas?
  - **Notification** of changed parts of data (“invalidation”, requires little bandwidth, good for small read-to-write ratio and large amount of data)
  - **State transfer**: Transfer the modified **data** (good for high read-to-write ratio); may transfer data, change logs, and/or aggregated
  - **Operation transfer**: Propagate the update operation (“active replication”, uses little bandwidth if parameter size is small, but requires more processing power and determinism)

## Pull (client)-based protocols

- A replica requests another replica to send it any updates it has at the moment
- Often used by client caches
- I.e. client polls server if updates are available
- E.g. Web “modified since”
- Response time increases in case of a cache miss
- Unicasting instead of multicasting

## Push (server)-based protocols

- Updates are propagated to other replicas without those replicas asking for updates
- Used by permanent and server-initiated replicas, but also by some client caches
- High degree of consistency (consistent data can be made available faster)
- If server keeps track of clients that have cached the data, we have a “**stateful**” server: limited scalability and less fault tolerant
- Often, multicasting is more efficient

## Pull versus Push Protocols

Assumption: single server and multiple clients with cache

Issue	Push-based	Pull-based
State of server	List of client replicas and caches	None
Messages sent	Update (and possibly fetch update later)	Poll and update
Response time at client	Immediate (or fetch-update time)	Fetch-update time

Remark: () denote cases where notifications are sent out.

## Hybrid: Lease

---

- ❑ How long should a stateful server keep track of a client?
- ❑ Leases **dynamically switch** push and pull
- ❑ A lease is a **promise** by the server to send updates to the client cache (i.e. to **push**) for a fixed period of time
- ❑ After **expiry**, client has to **poll** or request new lease (time can be adapted dynamically)
- ❑ Types of leases with dynamic time:
  - age-based (e.g. Web): last time modified
  - renewal-frequency: popularity of data for client
  - state-space: server overload reduces lease time

## Replication

---

- ❑ Basic concepts and consistency models
- ❑ Design considerations
- ❑ Replication protocols (DS)
- ❑ Replication in DB, SOA, P2P
- ❑ Replication and middleware

## Blocking vs. non-blocking

---

**When** are (push) updates propagated?

- ❑ Synchronous (**blocking**):  
All replicas are updated immediately, then reply to client
- ❑ Asynchronous (**non-blocking**):  
Update is applied on one copy, then reply to client, propagation to other replicas afterwards

## Replication Protocols

---

- ❑ Up to now: consistency models + general design considerations
- ❑ Implementation of the models? → replication protocols
- ❑ **Different protocols for different domains:**  
distributed object systems vs. databases vs. GRID vs. peer-to-peer systems (P2P) vs. SOA
- ❑ Concepts, commonalities, differences?

## Replication Protocols

What is replicated?

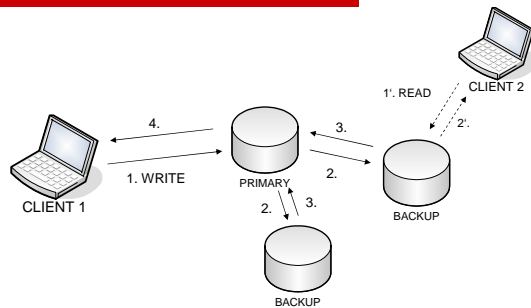
- ❑ “Traditional” Distributed Systems: objects, processes
- ❑ Database Systems: data (tables, rows)
- ❑ GRID: data (files, database)
- ❑ P2P: files, typically read-only
- ❑ Service-oriented system: services, stateful resources (data)

## DS Replication Protocols

**Classification:** Where are updates initiated? Is there a **primary** copy to which all write operations shall be forwarded?

- ❑ **Primary** replica: → Primary-based protocols
- ❑ **Group** of replicas: → Replicated-write protocols: active replication, quorum based protocols (voting)

## Synchronous Primary-backup



1. Client initiates write operation at primary
2. Primary processes operation and propagates updates to backups
3. Backups apply update and reply to primary
4. Primary replies to client

## Synchronous (blocking) replication

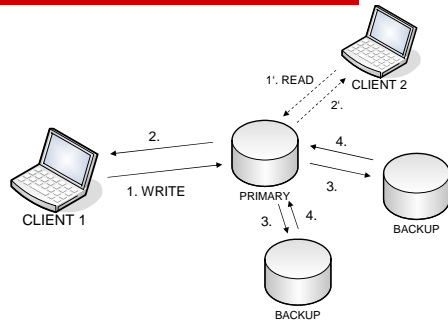
Advantages:

- ❑ No inconsistencies (identical copies)
- ❑ Reading the local copy yields the most up-to-date value
- ❑ Changes are atomic

Disadvantages:

- ❑ A write operation has to update all sites
  - slow
  - not resilient against network or node failure

## Asynchronous Primary-backup



1. Client initiates write operation at primary
2. Primary processes operation and replies to client
3. Primary propagates updates to backups
4. Backups apply update and reply to primary

## Asynchronous (non-blocking) replication

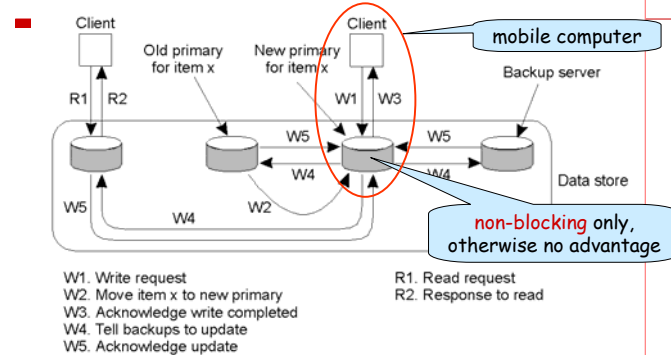
Advantages:

- ❑ Fast, since only primary replica is updated immediately
- ❑ Resilient against node and link failure

Disadvantages:

- ❑ Data **inconsistencies** can occur
- ❑ A local read does not always return the most up to date value

## Local-Write Protocols



- ❑ **Primary-backup** protocol in which the **primary copy migrates** to the process wanting to perform an update. Multiple, **successive update** operations can be carried out **locally**, while **reading** processes can still access their **local copy**.

## Primary-Backup (passive) Replication

Advantages:

- At least one node *exists* which has all updates
- Ordering guarantees are relatively easy to achieve (no inter-site synchronization necessary)
- Non-deterministic servers possible (in case of state-transfer only!)

Disadvantages:

- Primary is bottleneck and single point of failure
- High reconfiguration costs when primary fails

## Coordinator cohort replication

- ❑ Only one replica receives request (coordinator)
- ❑ However, coordinator may be **different** for different requests

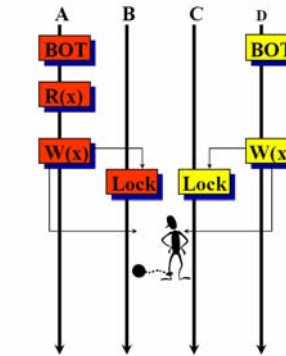
Advantages:

- ❑ No central bottleneck
- ❑ No single point of failure

Disadvantages:

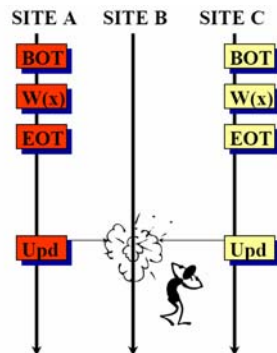
- ❑ Updates need **coordinated (distributed) concurrency control**
  - sync: distributed locking → **deadlocks** can occur
  - async: inconsistencies → **reconciliation**

## Deadlock in sync. coordinator cohort



©Gustavo Alonso, IKS, ETH Zürich

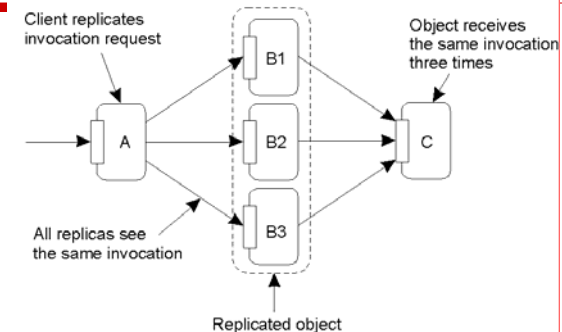
## Conflicts in async. coordinator cohort



©Gustavo Alonso, IKS, ETH Zürich

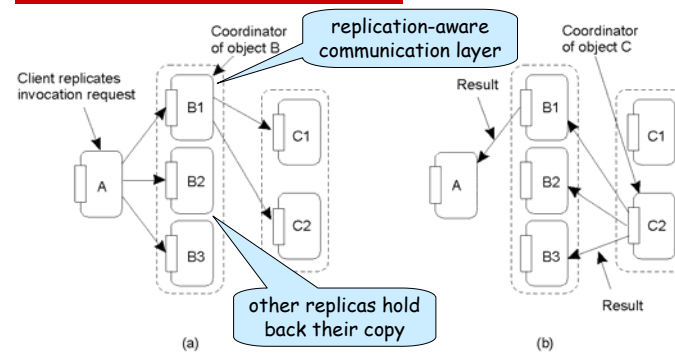
1. All transactions are executed **locally**
2. Changes are propagated to other replicas **after** transaction is executed
3. Conflicts can occur
4. Conflict resolution = **reconciliation**

## Active Replication



1. Operations need to be carried out in the same **order** everywhere (totally ordered multicast)
2. Replicated **invocations** (not a problem only of objects, but any client/server)

## Active Replication (2)



a) Forwarding an invocation request from a replicated object.

b) Returning a reply to a replicated object.

Sender- vs. receiver-based detection of multiple messages

## Active Replication

Advantages:

- Simplicity (same code everywhere)
- No bottleneck
- No single point of failure

Disadvantage:

- Determinism required
- Ordering more difficult

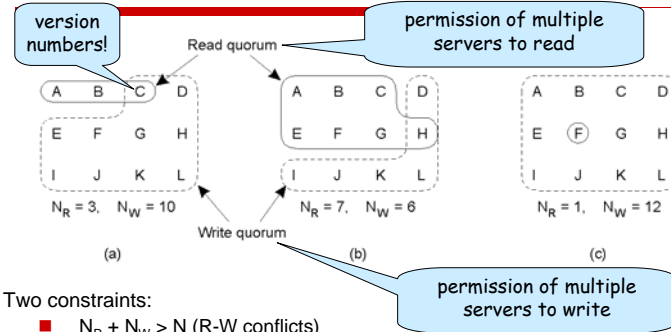
## Determinism

- Assumption: Consistent replicas at begin
- Determinism: when the **same operations** are applied in the **same order**, all replicas will produce the same result
- Reasons for non-determinism: random(), time(), multi-threading (thread scheduling unpredictable), transaction scheduling (concurrency control – timeouts for deadlock detection), ...

## Quorum-Based Protocols

- Write operations are performed on a write quorum  $N_W$  of replicas
- Read operations are performed on a read quorum  $N_R$  of replicas
- Two constraints must be obeyed:
  - $N_R + N_W > N$  (R-W conflicts)
  - $N_W > N/2$  (W-W conflicts)
- Version numbers needed
- "Majority voting"

## Quorum-Based Protocols



Two constraints:

- $N_R + N_W > N$  (R-W conflicts)
  - and  $N_W > N/2$  (W-W conflicts)
- a) A correct choice of read and write set
- b) A choice that may lead to write-write conflicts
- c) A correct choice, known as ROWA (read one, write all)

## Epidemic Protocols

- Eventual consistency
- Do not solve update conflicts (usually they do not occur or are easy to solve)
- As few messages as possible
- Model of spreading infectious diseases
  - Infective server: holds updated replica
  - Susceptible server: not yet updated
- Excellent scalability!
- Removing data is tricky (death certificates)

## Gossip Protocols

- Epidemic protocols with gossiping or “rumor spreading” as propagation model
- Analogous to real-life: P tries to push new update to Q, as long as several others already know, then it loses interest in spreading
- Good way of rapidly spreading updates
- However, some fraction of servers always remains ignorant → combining gossiping with anti-entropy will do the trick

## Replication

- Basic concepts and consistency models
- Design considerations
- Replication protocols (DS)
- Replication in DB, SOA, P2P
- Replication and middleware

## Database Replication Protocols

- ❑ Similar concepts as in distributed (object, process, file) systems
- ❑ BUT: subtle differences, different terms
  
- ❑ Eager = synchronous (blocking)
- ❑ Lazy = asynchronous (non-blocking)
- ❑ Primary copy = primary backup = passive replication: changes are initiated at the primary
- ❑ Update everywhere (~ coordinator-cohort): changes can be initiated at any replica

## DB Replication Protocols

### Primary-Backup (Primary-Copy):

- ❑ Conceptually equivalent to primary-backup replication in DS (objects, processes, ...)

### Update Everywhere:

- ❑ Conceptually equivalent to coordinator-cohort replication
- ❑ Updates can be initiated at any replica
- ❑ Advantages:
  - Any site can run a transaction
  - Load is evenly distributed
- ❑ Disadvantages:
  - Copies need to be synchronized

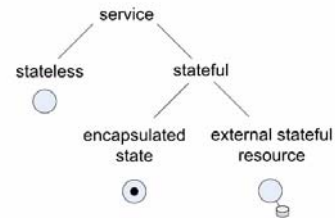
## Database Replication: Combinations

	Update Propagation	
Update Location	Eager Primary Copy	Lazy Primary Copy
	Eager Update Everywhere	Lazy Update Everywhere

## Replication in Service-oriented Systems

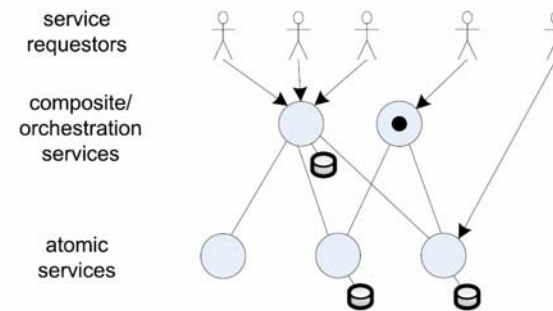
- ❑ What is a software service?
  - Computational element that performs some function
  - Accessible via (standardized) interface
  - Implementation not relevant to customer
  - Services can be composed
- ❑ Examples:
  - Atomic service: flight booking
  - Composite service: travel agency service combines flight booking, hotel booking and car rental services

## Stateless vs. Stateful Services

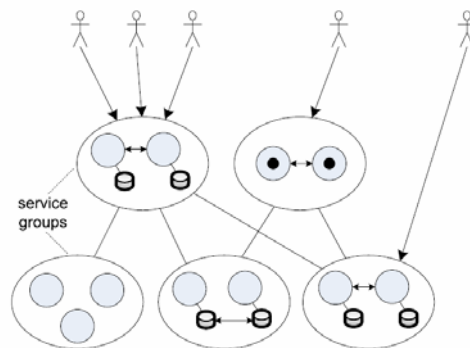


- Stateless service:
  - does not maintain state, e.g. file compression service
- Stateful service:
  - Encapsulates state or persists it in an external stateful resource (file, database), e.g. flight booking service

## Example for a Service-oriented System



## Replicated Service-Oriented System



## Service Replication: Conclusion

- Replication on service level is similar to replication of objects
  - Some differences: granularity, transaction model, etc.
- Replication via data store can use standard mechanisms of database or file management system

## Replication in P2P systems

---

- ❑ Distributed ownership → lack of centralized control, no global knowledge (scale!)
- ❑ Cooperation between nodes is limited → nodes go offline anytime without informing peers
- ❑ Node heterogeneity and link variations
- ❑ Mostly immutable data
- ❑ Legal issues across borders
- ❑ → Replication can only be **requested** and is associated with **uncertainty**
- ❑ → what is legal to replicate?

## Replication

---

- ❑ Basic concepts and consistency models
- ❑ Design considerations
- ❑ Replication protocols (DS)
- ❑ Replication in DB, SOA, P2P
- ❑ Replication and middleware

## P2P Replication Techniques

---

- ❑ Different terms used
- ❑ Active replication in P2P (~ push):
  - A peer actively attempts to replicate a data item to some other peer.
  - Concrete algorithms differ on
    - ❑ Number of replicas (e.g. uniform vs. proportional)
    - ❑ Location of replicas
- ❑ Passive replication in P2P (~ caching):
  - Peer shares downloaded data.
  - Intermediate nodes cache copies.

---

How can e.g. primary-backup replication be implemented?

Which middleware components are needed?

## Features of Replication Middleware

---

1. Interception of client calls
  - Ideally, client should not need to care about replication.
  - Client calls object as usual.
  - Middleware intercepts the call and triggers replication logic.→ Invocation Service

## Features of Replication Middleware (cont.)

---

2. Management of replicas
  - Middleware needs to know, where replicas are located.
  - Middleware needs to know the roles of replicas (primary vs. backups).
  - Middleware should be able to change the role of replicas (e.g., in case of the crash of a primary)→ Replication Manager

## Features of Replication Middleware (cont.)

---

3. Actual replication logic
  - Triggering of update propagation
  - Recovery
  - Reconfiguration
  - Reconciliation of divergent replicas in case of optimistic protocols→ Replication Protocol

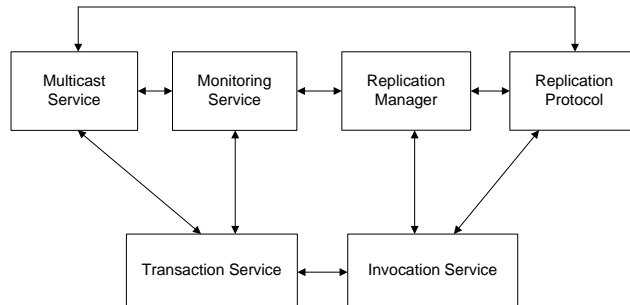
## Features of Replication Middleware (cont.)

---

4. Detection of node crashes and link failures
  - Reconfiguration of the protocol might be required→ Monitoring service
5. Reliable multicast of messages
  - Updates need to be propagated→ Multicast service
6. Support for transactions
  - Transaction service

## Replication Middleware Architecture

---



## Summary

---

- Replication helps to achieve better performance and fault tolerance
- Chosen replication protocol depends on different parameters: consistency requirements, read/write ratio, number of clients, etc.
- Most important protocols:
  - Primary-backup replication
  - Coordinator-cohort/Update-everywhere replication
  - Active replication
  - Quorum-based protocols
- Need to be adopted for domain:
  - distributed object system, file system, database system, service-oriented system, P2P system, etc.