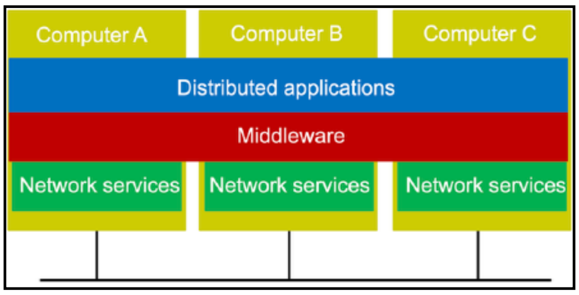
Middleware

0: Motivation

0.1 – Definiton

Why Middleware?

- In an enterprise application landscape different applications need to be integrated.
  > Enterprise Application Integration
- Middleware helps to connects and operate distributed applications. It is a layer between the applications and the network.
- Middleware Hides Heterogenity
  > Computer architecture, operating system, APIs



What is Middleware?

Middleware compromises services and abstractions that facilitate the design, development, and deployment of distributed applications in heterogeneous networked environments.

0.2 – Cooperation Models

Request/Reply

Adressing: indirect

Initiation: consumer

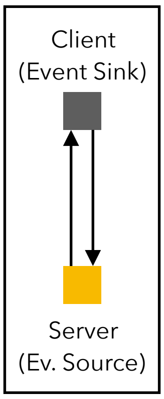
- Communication flow

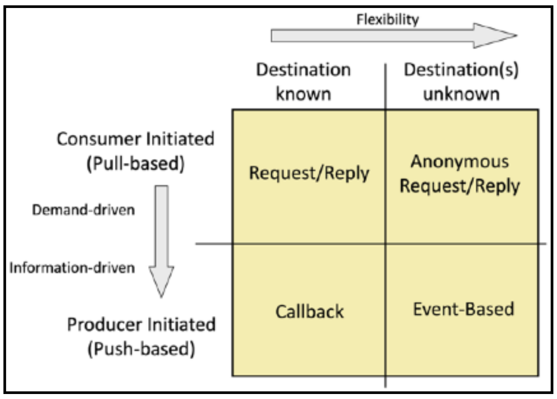
I. Client sends request to adressed server

II. Adressed server receives and processes request and issues reply back to the client

III. Client receives reply

- Evaluation
  > Client depends on the server's data and/or functionality
  > Server does not depend on the client





Anonymus Request/Reply

Adressing: indirect

Initiation: consumer

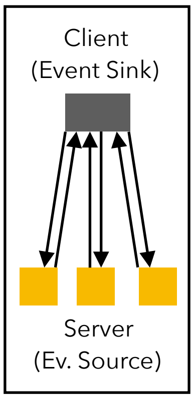
- Communication flow

I. Client issues request but dies not address any specific server

II. Appropriate server receive and process request and potentially issue a reply back to the client

III. Client receives and consolidates replies

- Evaluation
  > Loose coupling, easy exchange of server



Callback

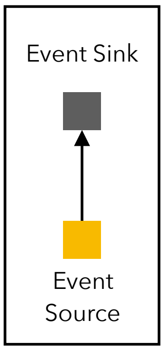
Adressing: direct

Initiation: producer

- Communication flow (consumers are registered at their dedicated producers)

I. Producer sends information to registeres consumers

- Evaluation
  > Producer can customize due to knowledge on consumer
  > High coupling



Event-Based

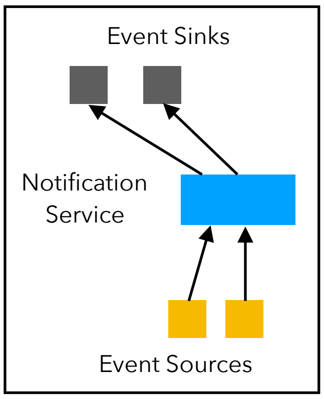
Adressing: indirect

Initiation: producer (consumers are subscribed to relevant events)

- Communication flow

I. Producers publish notification about event (state changes)

- Evaluation
  > Producers and consumers are decoupled (Notification Service = Middleware)
  > Requires more compley infrastructure

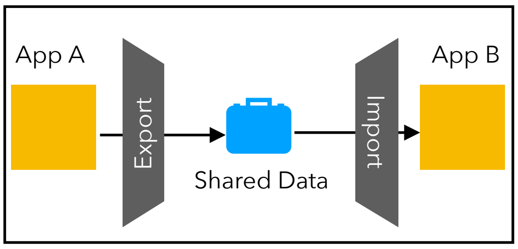


0.3 – Integration Pattern

File Transfer

Every application produces, shares and consumes files.
  > Integrators transform files into different formats
  > Applications produce files at regular intervals according to nature of business

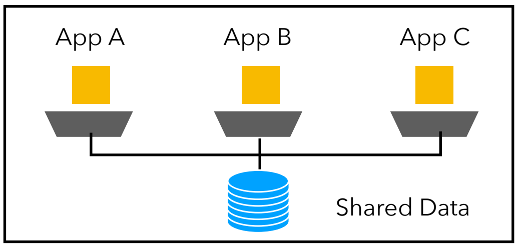
- Evaluation
  > (+) files are available on every operating system, integrators need no knowledge on actual application-workflow (low coupling)
  > (-) agreement on naming & directories, garbage collector needed, locking mechanism needed (parallel access), infrequent updates can lead to inconsistencies



Shared Database

Application store their data on a common database accessible to other applications.
  > Define schema of database to meet needs of the different applications

- Evaluation
  > (+) consistency, rely on common data and query model (SQL), concurrency
  > (-) unified schema needed > complex, schema is volatile due to updates by software vendors, possible performance bottleneck, deadlocks



Remote Procedure Call

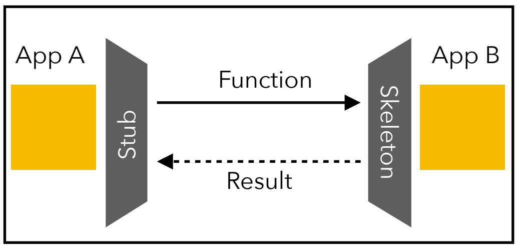
Application provide some of their procedures so that they can be invoked remotely, initiating behaviour and data exchange.

> Each application is object/component with encapsulated data

> Provide APIs

- Evaluation
  > (+) multiple interfaces on same data for different requesters, developers are used to this

  > (–) high coupling as every application needs to negotiate interfaces with its neigbours, performance issues due to remote procedure calls

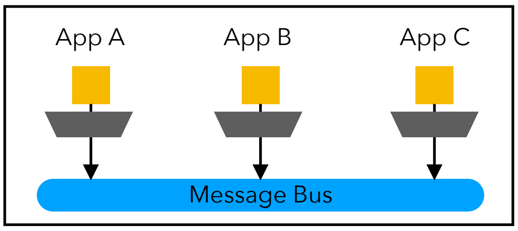


Messaging

Every application is connected to a common messaging system (used for data exchange and remote procedure calls).

- Evaluation
  > (+) Asynchronus communication > decoupling in space/time/control, seperates integration decisions from application development

  > (–) developers are not used to asynchronus design, testing/debugging harder, needs additional „glue-code" to fit everything together



1: Remote Procedure Calls

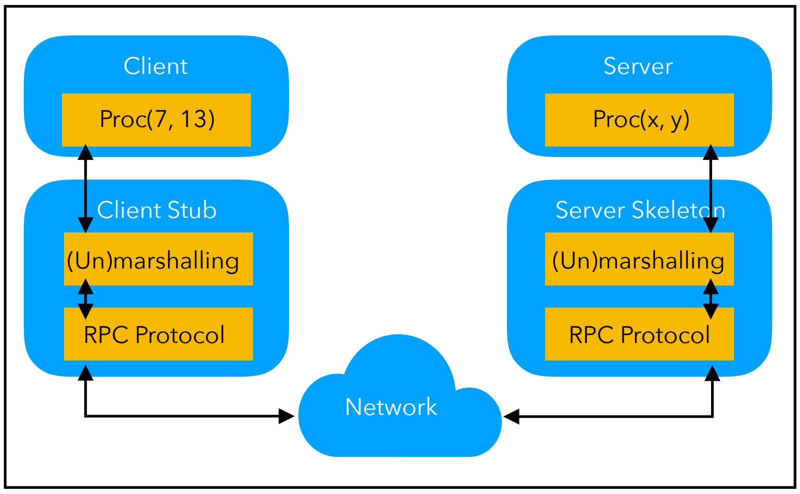
1.1 – Remote Procedure Calls

What are Remote Procedure Calls?
Application provide some of their procedure in a public API to initiate procedures & data exchange (hides network complexity > appears like local call).

RPC Communication

1. Client calls remote procedure (like local call); issued request to Client Stub
2. Parameters are encoded
3. Command is send to Server Skeleton by RPC
4. Server Skeleton receives RPC
5. Parameters are decoded
6. Server executed procedure

- Process vice-versa to send back procedure result.
- Call-by-Reference not working
  > Need for copying (serialize)

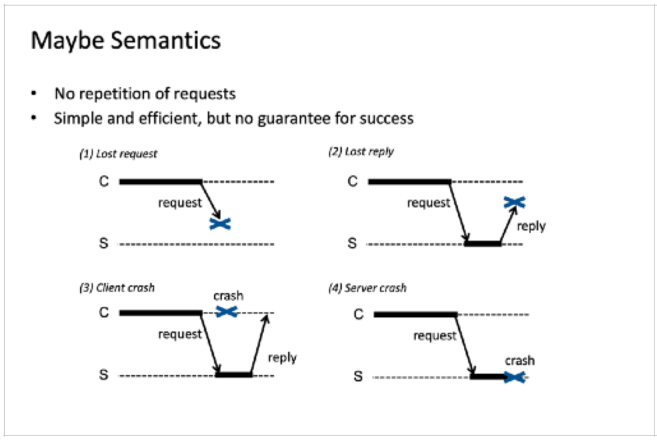


RPC Binding

- Static binding (request/reply): hardcoded reference
- Dynamic binding (anonym. request/reply): additional layer to locate server (name service)

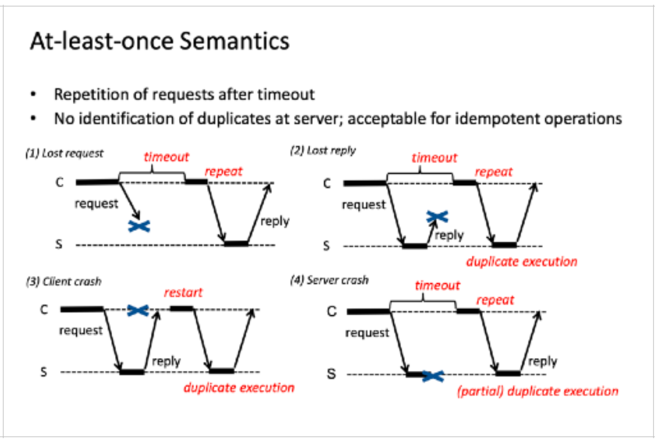
RPC Erros
Possible failures need to be handled (Network: lost request/reply, Client crash, Server crash).

- Recovery Capabilities (need depending on use-case)
  > maybe: none
  > at-least-once: time-out (c)
  > at-most-once: time-out (c), track status of calls (s)
  > exactly-once: time-out (c), track status of calls (s), tracking transaction system (s)
- Choose failure semantics based on call's properties (avoid unecessary overhead)



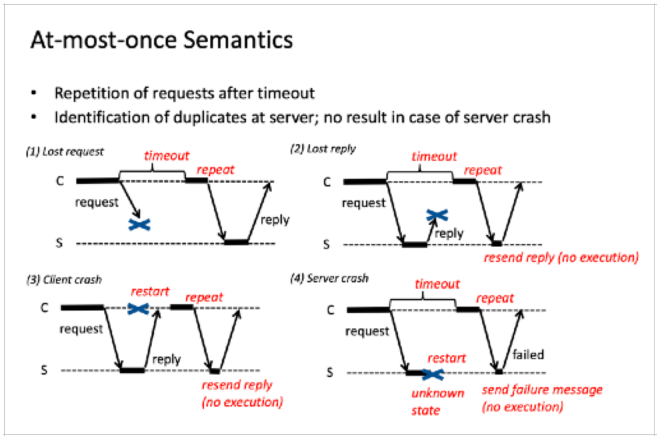
Maybe Semantics

- No repetition of requests
- Simple and efficient, but no guarantee for success



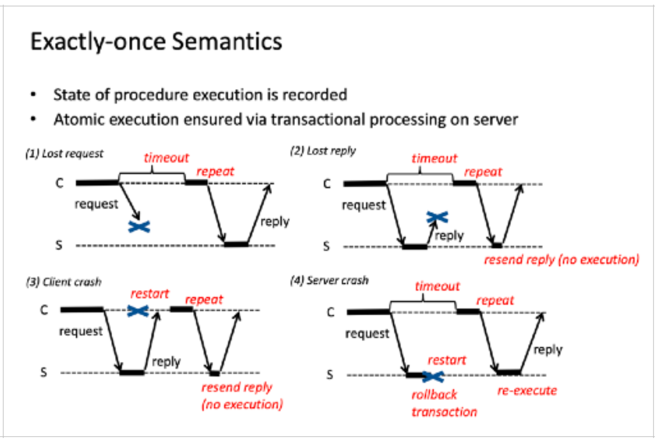
At-least-once Semantics

- Repetition of requests after timeout
- No identification of duplicates at server; acceptable for idempotent operations



At-most-once Semantics

- Repetition of requests after timeout
- Identification of duplicates at server; no result in case of server crash

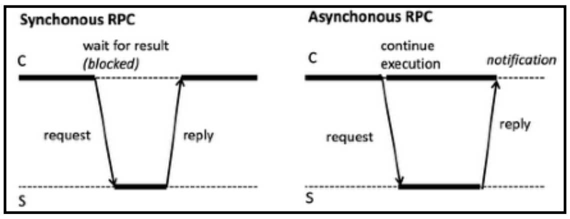


Exactly-once Semantics

- State of procedure execution is recorded
- Atomic execution ensured via transactional processing on server

Client-Side Implementation

- Decoupling of control flow at client (asynchronus RPC avoids potentially blocked threads)
  > Locks would appear frequently due to: Network latency, remote server overload, failures



Server-Side Implementation

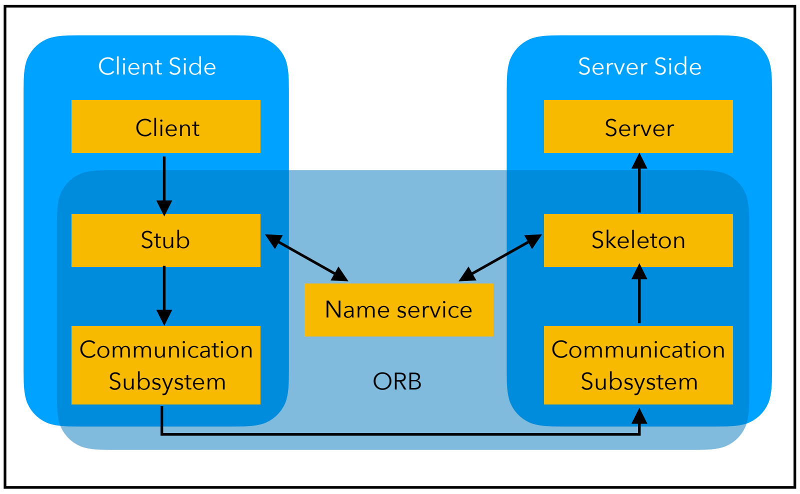
- Single-threaded server (insufficient usage of resources)
- One thread per request (pot. unlimited number of threads)
- Thread pool with scheduler: limited number of threads + input queue

$$N_{\text{threads}} = N_{\text{CPU}} \cdot U_{\text{CPU}} \cdot (1 + \frac{W}{C}) \quad U_{\text{CPU}} : \text{usage}, \quad W : \text{wait time}, \quad C : \text{compute time}$$

1.2 – Remote Method Invocation (RMI)

What is RMI?

In RMI objects in different processes communicate with each other. Remote interface show remotely accessible methods of an object.



RMI Communication

Object Request Broker (ORB): Middleware to execute remote calls.

- Identifies & locates remote objects
- Executes method calls
- Manages objects' lifecycle
- Binds client to server objects

Interface Definition Language (IDL)

- Purpose
  > Describe objects' interfaces of the objects being used by an application
  > Server as input for stub & skeleton generation
- No single format available; possible e.g. in Java, C#

Java RMI

- Interface of remote object: defined like Java Interface, must extend rava.rmi.Remote
- Stub class is generated from remote interface implementation using stub generator (via rmic)
- Any class class implementing a remote object must create a stub class (used as reference for remote object)

1. Interface:
```
public interface Hello extends java.rmi.Remote {
   String sayHello() throws RemoteException;
}
```
2. Implementation of the interface:
```
class HelloImpl extends UniCastRemoteObject implements Hello {
   HelloImpl() throws Remote Exception{
   };
   public String sayHello() throws RemoteException {
      return „Hello World!“;
   }
```
3. Generating Stub Class:
   > Compile interface implementation
```
javac HelloImpl.java
```
   > Generate stub class by compiling with rmic
```
rmic HelloImpl → HelloImpl_Stub.class
```
4. Server Program
```
public class Server {
   public static void main(…) {
      //sth
      Naming.rebind(„jrmi://„+regHost+“/helloobj“, new HelloImpl());
      System.out.println(„Hello Server ready!“);
   }
}
```
   > At instantiation of HelloImpl via constructor skeleton and stub objects are instantiated, too
   > Stub serves as reference to object (reference to stub is given by constructor)
5. Client Program
```
public class Client {
   public static void main(…) {
      //sth
      Hello obj = (Hello) Naming.lookup(„jrmi://„+regHost+“/helloobj“);
      System.out.println(obj.sayHello());
   }
}
```
   > Lookup in naming service via symbolic name (helloobj), return stub for target object
6. Naming Services in Java
   > Two possibilities: RMI registry (easy), JNDI service
      1) Server Program creates the RMI registry
```
try {
   LocateRegistry.createRegistry(Registry.REGISTRY_PORT);
}
catch (RemoteException e) { /* handle exception */ }
```
      2) Program rmiregistry started on server
```
$ rmiregistry &
```
   > Client needs to know host name & port of registry to connect

1.3 – Representational State Transfer (REST)

What is REST?

- request/reply styled interaction, alternative to RPC/RMI
  > No sessions (connections) between requests
  > No functional API, based on states (CRUD: create, read, update, delete)
- Server sends representation/state of a resource & manipulates it (based on HTTP commands)
  > POST: create new (sub)-resource below the specified resource
  > PUT: create specified resource (if existing > modify)
  > PATCH: update partial resource
  > GET: requests resource form server
  > DELETE: deltes specified resource

REST vs. RPC

- REST: data-centric → define resources → usage: resources & entities
- RPC: procedure-centric → define operations → usage: actions (send messages, etc.)

2: Messaging

2.1 – Basic Concept

Messaging

- Using message channels to transfer messages
- Problems to handle: routing, format, connecting endpoints to application (adapters)

Types of Message Channels

- Message channel: one application writing & one application reading from a channel
- Point-to-point channel: exactly one receiver will receive one message
  > Easy load balancing by sending application (just send on different channels)
- Publish/Subscribe: message send to all interested subscribers
  > One input duplicated to multiple (subscribed) outputs (each receiving exactly one)
- Invalid message channel: receiver moves inpropper messagages > diagnosis
- Dead letter channel: messages, which cannot be delivered/are expired > diagnosis
- Guaranted delivery channel: message deliverey guaranted in case of system fail
  > Store-and-forward principle: messages are stored on safe hard-disk before sending
  > Lowering performance

Channel Adapter

Connecting application and message system (on different layers/API).

Messaging Bridge

Connecting different messaging systems (with different protocols)
  > set of channel adapters between messaging systems

Message Bus

Seperate applications use one generic bus
- Combination of: canonical data model, common command set, messaging infrastructure

Composition of Messages

1. Header: describes data (origin, destination, content, etc) < relevant to messaging system
2. Body: actual data meant for applications < relevant to applications

Basic Concepts (Issues)

- Message intent: what sender expects receiver to do
- Message response: response/confirmation expected?
- Huge amounts of data: typically not fitting into a single message
- Slow messages: sender does not know transmission time

Types of Messages

- Command messages: invoke procedure in different application (looses coupling)
- Document messages: pass set of data to other application (timing less important)
- Event messages: event notification (event: instanteous & asynchronus state transition) real-time
  > Push model: message combines document+event message
  > Pull model: event message to notify observer, obs. sends request, obs. gets state reply
- Request/Reply messages: pair of request/reply messages on seperate channels
  > Pattern: request via point-to-point/publish-subscribe, reply via point-to-point
  > Return adress should be part of request
  > Message should feature unique correlation identifier to match requests and replys
- Message sequences: used for larger amounts of data, which cannot be send in one message
  > Sequence identifier: distinguishes sequences
  > Position identifier: orders message of a sequence & unique identifier
  > Size/End idicator: marks sequence’s end & number of messages
- Slow messages: expiration time → *if*(expired) *then* DeadLetterChannel

---

2.2 – Practical Use

Java Messaging Service (JMS)

- Independent standard for asynchronus point-to-point & publish/subscribe messaging
  > point-to-point: adressed to specific destination queues (message channels), each message has one consumer, queues retain messages until successfully processed (guaranteed deliv.)
  > publish/subscribe: messages are published to topics multiple consumers can subscribe to

JMS Message Composition

- Header: immutable header properties (e.g. messageID, timestamp)
- Properties: mutable key-value properties (e.g. state description)
  > Optional fields for header edited while sending, only
- Body: one of five defined message types
  > Message: emtpy, only header & properties
  > StreamMessage: stream of Java primitive values
  > MapMessage: set of key-value pairs
  > TextMessage: String (e.g. XML-doc)
  > ObjectMessage: serialized Java object
  > BytesMessage: stream of uninterpretet bytes

3: Data Represenation

3.1 – Message Translator

Message Translator

- The message translator translated messages from different systems using different formats to ensure a stable communication.
- Messaging Systems often use a common internal standard to decrease coupling (exceptions for real-time/high-performance services)

Levels of Transformation

Layer	Deals with	Transformation Needs	Tools & Techniques
Data Structures (Application Layer)	Entities, associations, cardinality	Aggregate many-to-many relation into one field	Structural mapping patterns, ER diagrams, class diagrams
Data Types	Field names, data types, value domains, constraints, code values	Convert ZIP code from numeric to String	EAI visual transformation editors, XSL, database lookups, custom code
Data Represenation (Syntax Layer)	Data formats (XML, JSON, protofuf, etc.); Character Sets (ASCII, UTF-8); encryption, compression	Parse data representation and render in a different format; de/encrypt	XML parsers, EAI parser/renderer tools, custom APIs
Transport	Communication protocols (TCP/IP sockets, HTTP, etc.)	Move data across protocols (w/o affecting content)	Channel Adapter pattern, EAI adapters

Canconical Data Model

Common Data Format for messages to decrease number of needed adapters & coupling.
> Trade-off between maintainability & performance

3.2 – Data Formats

Binary Formats

- External Data Represenation (XDR)
  > 4-byte blocks serialized in big-endian order (n bytes data, r bytes padding; n+r mod 4 = 0)
  > Data types not encoded in binary (implicit typing)
- Abstract Syntax Notation One (ASN.1)
  > Platform independent description of data-types
  > Byte streams [tag, length, value]
- Java Object Serialization (JOS)
  > Stream based transmission of serialized java objects implementing `java.util.Serializable`
  > Lots of meta data, only applicable in java environment
- Protobuf
  > More efficient binary encoding than ASN.1 (by zig-zag encoding, variable length,

Text Formats

- Extensible Markup Language (XML)
  > Widely applicable standard for data exchanged
  > Human readable plain text, large overhead
- JavaScript Object Notation (JSON)
  > Language independent format (easy integration)
  > Less verbose than XML, still large overhead, human readable

4: Matching and Routing

4.1 – PubSub

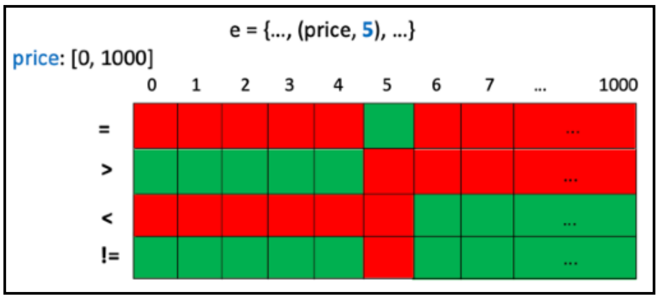
Routing
Directing messages to propper receiver.

Events
Events in form of messages that are filtered against certain queries (matching).

Matching
Given an event e, a set of subsriptions S > determine all subscriptions $s \in S$ that match e.
- Channel-Based Matching: channels categorize events, subscribers subscribe to channels
- Topic-Based Matching: subjects are categorized hierarchically in a tree-structure
- Content-Based Matching: filter (logical expressions (key, operator, value)) evaluating the messages content (high decoupling, pot. performance issues)

4.2 – Algorithms for Content-Based Matching

1: Predicate Matching
Given a set of predicated and an event e, identify all predicates $p \in P$ evaluating to true under e resulting in a predicate bit vector.
- Top-Level Data Structure: hash-table on attribute name (key)
- General-Purpose Data Structure: ordered list for each operator (key, value)
- Specialized Matching for finite domains: Table based matching


e = {..., (price, 5), ...}

price: [0, 1000]

	0	1	2	3	4	5	6	7	... 1000
=	🟥	🟥	🟥	🟥	🟥	🟩	🟥	🟥	🟥 ...
>	🟩	🟩	🟩	🟩	🟩	🟥	🟥	🟥	🟥 ...
<	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩 ...
!=	🟩	🟩	🟩	🟩	🟩	🟥	🟩	🟩	🟩 ...

2: Subscription Matching
- Counting Algorithm: count number of satified predicates for each subscription and compare with total number of predicates

4.3 – Distributed PubSub: Routing

Overview
- Broker = service instances: each broker manages a set of local clients (comp. router), each local client is attached to exactly one broker
- Events are forwarded stepwise through broker network
- Requirements: Correctness & Accuracy, Performance, Scalability(, prevent cycles)

Techniques

- Flooding: each event is delivered to all brokers, borker forwards to
  - > all neighbours if received from local client
  - > all neighbouring brokers except for delivering brokers
- Content-Based routing: each broker manages filter-based routing table
  - > Rounting entries: $(F, D) \rightarrow$ Filter, Destination
  - > Forwards to all entries evaluating to true, except for delivering broker

Routing Algorithms

Rounting algorithm is needed to keep routing entries for content-based routing updated.

- Simple Routing: each subscription added to every routing table, entries are spread via flooding
- Identity-Based Routing: uses identity test > identical filters are not forwarded
- Covering-Based Routing: if new filter is subset of existing > filter is not forwarded
- Merging-Based Routing: merging of files
  - > perfectly: $E(F) = E(G) \cup E(H)$
  - > imperfectly: $E(F) \supset E(G) \cup E(H)$
- Second routing table for forwarding the subscriptions

5: Management of Messaging Systems

5.1 – Managenent of Messaging Systems

Challenges
High rate of message throughput, potential failures and distributed architecture may be challenging. To ensure a working messaging system, we do
- Monitoring and controlling > logging
- Obersving and analyzing message traffic
- Testing and debugging

5.2 – Monitoring and Control

Control Bus
Seperated message bus for logging- and control-messages, such as
> configurations messages: set parameters, etc.
> heartbeat messages: periodic messages for validating
> test-messages: run tests & validate outputs
> exceptions: alert operator about misbehaviour
> statistics: statistics on current performance
> live console: displays general health style of messaging systems

Detour
Seperates channels in two channels (normal delivery, additional processing). The Detour-Router is instructed via the control bus.
- Simultaneous swtiching of multiple detour-routers is possible in a pub/sub architecture

5.3 – Observing and Analyzing Message Traffic

Observing and Analyzing Message Traffic
Tracking messages passing a channel provides information on potential misbehaviour. This should not interfere with the normal flow of the messaging middleware.

Wire Tap
Tap main-channel and send a copy to secondary channel.
- Evaluation: minimal interference with system (+), latency due to routing through wire tap (–)

Message History
Attach a list of traversed applications/components the message passed.
- Evaluation: easy approach, just add ID (+), history attached to message, which is lost after consumption (–)

Message Store
Whenever message is send, a duplicate is send to message store via control bus.
- Evaluation: easy (+), limited storage (–), increasing network traffic (–)

Smart Proxy

1. Smart proxy intercepts message on request channel, stores ID & return adress, modifies return adress to its listening channel
2. Message is forwarded to orginial target, processed, returned to Smart proxy
3. Smart proxy analyzes reply as desired, update return adress with original return adress from storage, forward unmodified message to original return adress

5.4 – Testing and Debugging

Testing and Debugging

Control bus describes a number of approaches to monitor health of message processing system.

Test Message

Injecting messages into message stream and confirm health of systems by checking the reply.

> Test data generator: generates test message
> Test message injector: inserts tests message into regular message stream
> Test message separator: extracts processed test message from regular output stream
> Test data verifier: compares actual & expected results

- Evaluation: deeper level of testing (+), overhead & additional load (–)

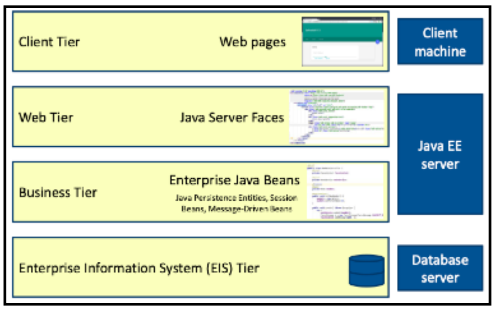
6: Application-Oriented Middleware

6.1 – Application Servers

Definition

Product in middle-tier of server-centric architecture providing middleware-services.

> e.g. Web service hosting, Deployment services, monitoring/logging, caching



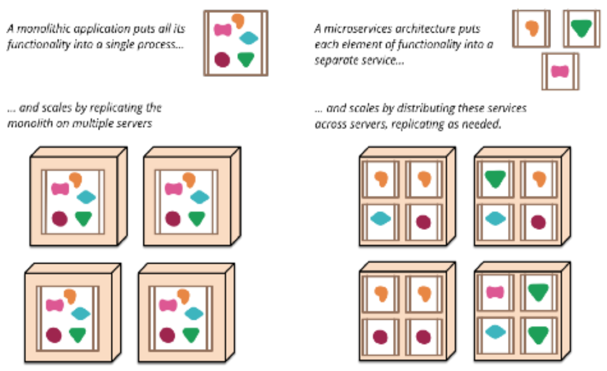
Java Enterprise Edition (Java EE)

Set of specifications of APIs and their interfaces for generating web pages, transactional queries on data bases, managing distributed queues, etc.

6.2 – Microservices

Definition

Microservies are an architectual style: developing applications as a suite of small services.


A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

Characteristics

1. Componentization of Services: services are out of process components rather than in-memory function calls
2. Organized arround Business Capabilities (rather than technology-oriented teams)
   > Conways Law: organisations designing a systems produce a structure, which is a copy of their organization's structure
3. Products, not Projects: not just developing; you build it, you run it
4. Smart Endpoints & Dump Pipes: decoupled cohesive applications, act like Unix filters
5. Decentralized Governance: no standardized technology platform, use right thing for right job
6. Decentralized Data Management: each service with its own database
   > cosistency is challenging
7. Infrastructure Automatization: continious delivery (often applied for microservices)
8. Design for Failure: failure of a service is highly likely > must be handled
9. Evolutionary Design: control changes in single services without slowing down overall change

6.3 – Web Services

Service-Oriented Architecture

SOA represents business activity, is self-contained, is a black box for its consumers & may consists of other underlying services.

Web Services

Web services connect multiply components through the internet using standardized protocols.

- Simple Object Access Protocol (SOAP) > pasing XML-encoded data, {Evelope, Header, Body}
- Web Services Description Language (WSDL) > describes web services (how to invoke, etc.)
- Universal Description, Discovery and Integration (UDDI) > dynamically find other web services

Web Services and Middleware

Web service infrastructure relies on middleware on application and communication level.

- Evaluation: standardizes (+), overhead & inefficiency (–)

6.4 – Business Process Management

Business Processes

Interaction of different services are considered a business process. (Web) services interact to implement a business process.

> Service Orchestration: *local* perspective, describes one party's behavior
> Service Choregraphy: *global* perspective, decribes global interactions among all involved parties

Business Process Execution Language (BPEL)

- Standard for service orchestartion
- XML-based

Activity	Description	Activity	Description
Sequence	Sequence of Activities	Scope	Fault/Exception Handlers
If	Supports XQuery as condition	compensate Scope	Used with catch or compensation handler elements
While	Loop, supports XQuery as condition	Rethrow	Only used as an activity on a fault handler
repeatUntil	do-while Loop, supports XQuery as condition	Compen-sate	In case of fault, a compensation action can be triggered
forEach	Loop, supports XQuery as condition	Validate	Validates XML data, can be used as an option for assign activity
Pick	Specifies a process to be executed according to received event	extension Activity	Custom activity implementation (has to be supported/deployed in process engine)
Flow	Parallel execution of processes		

7: Naming and Coordination in Distributed Systems

7.1 – Overview

Application of Naming

Naming and directory services are essential for distributed systems (name > physical address).

- Identification of an adress or attribute for a name (DNS)
- Identification of a machine for a service (RPC)
- Identification of a machine for an object (RMI)

7.2 – Domain Name System (DNS)

Domain Name System

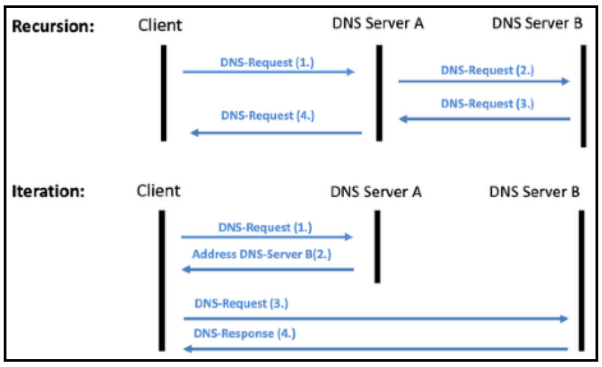
Resolves domains to physical IP-addresses based on worldwide distributed database of nameservers.

- Root domain: top of tree, unnamed level [(.)]
- Top level domain: indicates a country/region/type of organization [.de]
- Second level domain: names registered by an organization [tum.de]
- Subdomain: additional name, an organization can create [in.tum.de]

Name Server

Name servers resolute domains to their respective physical addresses.

- Authoritative name server: responsible for domain (max. one for each domain)
- Non-Authoritative name server: receives information from other name servers and answers requests by forwarding or loading cached results



7.3 – Lightweight Directory Access Protocol (LDAP)

Directory Service

Customizable information store (distributed) for users to locate resources and services.
LDAP is a standardized network protocol for querying and updating information in dir. service.

7.4 – Coordination in Distributed Systems

Leader Election

In a set of distributed processes/servers one process should become the leader, the others followers. Some Conditions shall be met:

- Termination: select leader within finite time
- Uniqueness: only one leader at a time
- Agreement: All processes are informed properly about the current leader

Fault Tolerance
Fault-tolerance can be ensured by passive replication of the current state on all followers.

Mutual Exclusion
Leader controlls access to a critical section (only one follower in critical section at any time).

Multithreading with Synchronization Barrier
Whenever multiple proceses need to be synchronized, a synchronization barrier has to be established. It prevents single processes to pursue until all processes are registered in a shared datastructure (e.g. tree in Apache ZooKeeper).

7.5 – Apache ZooKeeper

What is Apache ZooKeeper?
Open source project for highly reliable distributed coordination (incl. maintaining distributed cofiguration information, providing distributed synchronization, group services).
ZooKeeper Guarantees:
- Sequential Consistency: updates from clients will be applied in the order they were sent
- Atomicity: Updates either succeed or fail - no partial results
- Single System Image: only one server is visible to the world
- Reliability: applied updates are persistent until further changes are applied
- Timeliness: clients view of the system is up-to-date (within a time bound)

Data Model
Namespace ist similar to a file system (tree structure), where nodes can contain data & children.
- Each node is associated with an Access Control List.
- Clients can create watches on Znodes

API

`create`	`delete`	`exists`	`get data`
Creates node at a location in the tree	Deletes a node	Tests, if a node exists at a location	Reads the data from a node
`set data`	`get children`	`sync`	
Writes data to a node	Retrieves a list of children of a node	Waits for data to be propagated	

ZooKeeper Server Components
- Request Database: database containing the entire data tree
- Atomic Broadcast: ensures same order of updates on each server, if it fails, all servers reset
  Typical Communication
  > one folower requests update (enqued)
  > leader makes proposal and requests followers to acknowledge
  > followers send back ACK proposals
  > After majority send acknowledgement, leaders decides to commit
  > followers execute leaders decision

8: Standardization

8.1 – Standardization

What is a Standard?
A level of quality attainment. Something used as a measure, norm or model in comparative evaluations.

Benefits of Standardization
- Safety and reliability
- Support of government policies and legislation
- Interoperatbility (betweeen systems)
- Business benefits: market access, economies of scale, innovation, awareness
- Consumer choice

Issues of Standardizations
- Over- and under specification
- Features left out as consensus was not reached
- Agreement process may be to long
- Proliferations of standards, overlapping standards

8.2 – Case Study Standardization: CORBA

Storyline
CORBA (Common Object Request Broker Architecture) is a standard for RMI middleware. It worked until the rise of the internet, when adoption of new guidelines was to slow to compete with emerging other standards/protocols like HTTP and EJB (Enterprise Java Beans). Guidelines where adopted in a democratic and slow process by a board of more and less qualified experts.

Problems in Standardization Process
1. No required qualifications to participate in process
2. RFPs (request[s] for proposals) often call for unproven technology
3. Vendor respons to RFPs even when they have (known) technical flaws
4. Vendors have a conflict of interests when it comes to standardization
5. RFPs are often to complex as many features a merged into a single standard
6. Overseeing organ did not require a reference implementation in adoption process

8.3 – Strategies

Timing for Adoption

Early Adoption	Influence standard, early to market, better experience	Risk of failure, potential changes, lack of support
Late Adoption	Maturity of standard, support	No influence, no innovation

9: Queuing Theory

9.1 – Queueing Theory

Motivation

As number of requests are unknown & fluctuating, rate of requests and service time need to be estimated in order to build stable system, which are capable of handling request peaks by queuing the requests.

- How many service instances are needed to ensure stability?
- How many service instances are needed to guarantee a certain response time?

Queuing Theory

Queuing Theory provides meaningful estimations to answer the questions given above.

Performance Metrics

- Mean waiting time
- Server utilization
  > time-proportion, the server is busy: server utilization = mean arrival-rate · mean service-time
- Throughput
  > average number of completed jobs per unit of time, maximum throughput as measurement
- Average number of customers waiting
- Distribution of the number of waiting customers

Stochastics in Queueing

1. arrival-process: inter-arrival time is modelled as a probabilistic distribution
   > Arrival rate: $\lambda = \dfrac{1}{\text{mean inter-arrival time}}$
2. service-process: service time per Customer is modelled as a probabilistic distribution

These proccesses can be

- M (Markov): Exponential probability density
- D (Deterministic): All customers have the same value
- G (General): Any arbitrary probability distribution

Properties of Queueing Systems

- Calling population: how many possible calls $[0,\infty)$
- Queue capacity: how many requests can be enqueued $[0,\infty)$
- Service discipline: how are requests dequeued [FIFO, LIFO, Prioritized, Randomized]

Stability Conditions

A queue is stable, when it does not grow to infity over time.

$$\text{mean service time} < \text{mean inter-arrival time} \Longrightarrow \text{stable Queue}$$

9.2 – Analysis of Queueing Systems

M/M/1 Queue

- M: exponentially distirbuted inter-arrival time
- M: exponentially distributed service-times
- 1 Single Server Queue
- Properties (Standard Assumptions)
  > Infinite calling population
  > Infitine queue capacity
  > FIFO service discipline

Calculations

$\rho = \lambda \div \mu$	traffic intensity (occupancy), busy time
$P(n) = \rho^n \cdot (1-\rho)$	number of customers in the system
$P(0) = 1 - \rho$	empty system, idle time

μ average service rate λ average arrival rate, throughput

$N = \dfrac{\rho}{1-\rho}$	Mean number of customers in the system
$T = \dfrac{1}{\mu - \lambda}$	Total waiting time (incl. service time)

Exponential Distribution

$f(x) = \lambda \cdot e^{-\lambda \cdot x}$ where λ is the arrival/service rate

$\mu = 1 \div \lambda$

Poisson distribution

- Describes the number of arrivals per unit of time, if inter-arrival time is exponential
- Individuals in the population are independent

$$P_n(t) = \frac{(\lambda \cdot t)^n \cdot e^{-\lambda \cdot t}}{n!}$$

> Probability of seeing n arrivals in a period from 0 to t
> t is used to define the interval $[0,t]$
> n is the total number of arrivals in the interval $[0,t]$
> λ is the total avergae arrival rate in arrivals/sec

9.3 – Little's Law

Little's Law in Words

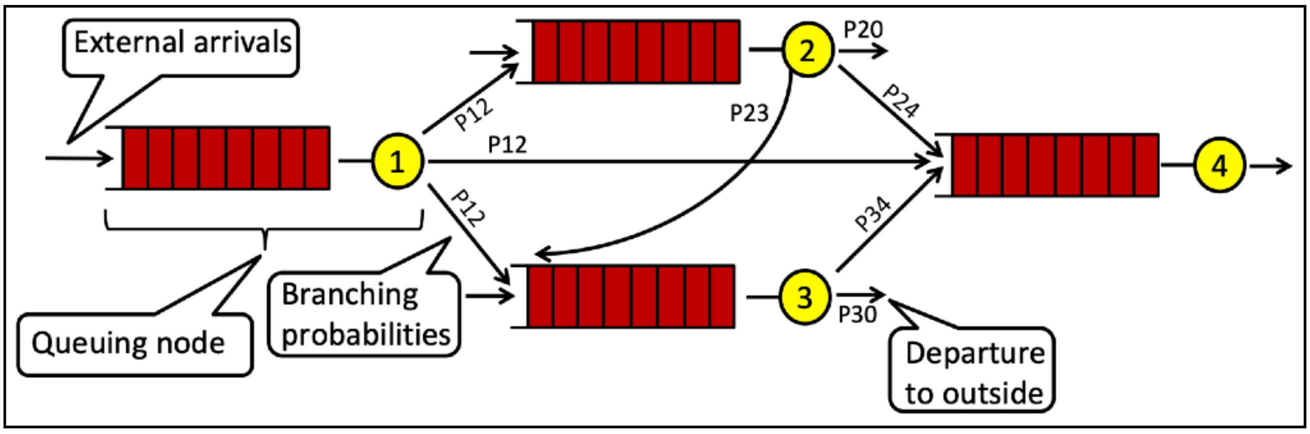
The long-term average number of customers in a stable system (L) is equal to the long-term average effective arrival rate (λ), multiplied by the averagy time a customer spends in the system (W). Therefore it does not depend on arrival/service distribution. $L = \lambda \cdot W$

$L = \dfrac{\lambda}{\mu - \lambda}$ Mean number of customers $W = \dfrac{1}{\mu - \lambda}$ Mean wainting time

9.4 – Analysis of Queueing Networks

Queuing Networks

Queues can be linked together to form a network of queues which reflect the flow of customers through a number of different service stations.



Mean Arrival Rates at a Node

$$\lambda_i = \gamma_i + \sum_{j=1}^{m} p_{ji} \cdot \lambda_i$$

m	number of nodes
λ_i	mean arrival rate into node i
γ_i	the external arrival rate
p_{ij}	branching probability from j to i

–

10: Excersises

10.1 – Modelling with Middleware Components

Cooperation Models

Classify Examples into their cooperation model. Consumers and Producers are given explicitly.

Integration Patterns

Classify the best-fitting integration pattern for given use cases. Reason the decision using advantages and disadvatages of the choosen pattern.

10.2 – Remote Procedure Calls

RPC Execution

Note different flows of RPC communications between different clients and servers. Each step is denoted as follows: $c \rightarrow s$ (msg), $s \circlearrowright$ (msg), $s \rightarrow c$ (msg)

RPC Failure Semantics

Note different flows of RPC communications, where servers and clients implement different semantical approaches to failure. Failures can appear at different points.

$\cancel{s}$	server crash	$s\,*$	server restart
$c \not\rightarrow s$ (msg)	Message loss	$s\cancel{\emptyset}$(msg)	proccessing error
$c \rightarrow s\,(e)$	Exception message	timeout	timeout, resend

- Maybe semantics:
  > Message loss: $c \not\rightarrow s$ (msg), $\square$
  > Processing error: $c \rightarrow s$ (msg), $s\cancel{\emptyset}$(msg), $\square$
  > Server crash: $c \rightarrow s$ (msg), $\cancel{s}$, $s\,*$ $\square$
  > Client crash $c \rightarrow s$ (msg), $\cancel{c}$, $s \rightarrow c$ (msg), $c\,*$, $\square$
- At-Most-Once semantics: one try, potential timeout
  > similar to maybe
  > Timeout: $c \rightarrow s$ (msg), $s \circlearrowright$ (msg), timeout, $\square$
- At-Least-Once semantics: repetition until no failure occurs
  > Message loss: $c \not\rightarrow s$ (msg), timeout, standard Communication, $\square$;
  $c \rightarrow s$ (msg), $s \circlearrowright$ (msg), $s \not\rightarrow c$ (msg), timeout, $c \rightarrow s$ (msg), $s \circlearrowright$ (msg), $s \rightarrow c$ (msg) $\square$
  > Processing error: $c \rightarrow s$ (msg), $s\cancel{\emptyset}$(msg), timeout, standard Communication, $\square$
  > Server crash: $c \rightarrow s$ (msg), $\cancel{s}$, $s\,*$,timeout, standard Communication, $\square$
  > Client crash: similar to server crash
- Exactly-Once semantics: just one execution, further requests are ignored

REST

Edit a Web-Application using the known CRUD REST commands.
- Content Types: application/json (POST, PUT, (PATCH), DELETE), text/plain (GET)
- GET/DELETE: add query behind questionmark, e.g. shop.com/items?price_net[gt/lte]=100
- PUT (for updates): specific domain adressing resource, new content in put-message

10.3 - Messaging

Message Types
Classify the best-fitting message type for given use cases. Reason the decision using advantages and disadvatages of the choosen type.

Messaging R/R with queued transactions
Request and Reply queues are used to ensure exactly-once semantics.

Nochmal nachfragen

10.4 - Pub/Sub I

Content-Based Matching
Enter the given predicates into lists for each attribute and its associated binaryoperator. The entries to the list are given as tuples of a value and a predicate ID.
A vector for each predicate is made, containing those servers, who demand for that predicate in their subscription. A second vector lists the maximum subscriber hits, beeing the number of appearences of a server in the predicate vector.
While Routing a message, a ‚hit count' vector is used to evaluate, wether a message needs to be redirected.

Content-Based Routing
Subscribptions are rerouted in the system and entered into routing rables containting subscriptions and publications.
Note the message flow through the network in the following notation:

- Message, which is not forwarded: $\{\emptyset\} \rightarrow \emptyset$
- Message, which is forwarded: $\{p_n, b_n, \ldots, s_n\} \rightarrow \emptyset$
- Message, which is forwarded and split up: $\{p_n, b_n, \ldots, \{b_m 1, \ldots, s_m\} \,||\, \{b_m, \ldots, s_m\}\} \rightarrow \emptyset$

10.5 - Pub/Sub II

Covering Filters
To minimize the overhead of checking subscription, merging algorithms are used.

Nochmal nachfragen - System?

Perfect Covering Filters
Create perfect covering filters from a set of input predicates. Note the perfect covering filter as its argument set and try to use less predicates than initially provided.

Imperfect Covering Filters
Create covering filters which may cover more than intended but thereby lowering the amount of needed predicates. Draft the set of additional covered values for each variable.

10.6 – Middleware and Orchestration

State Machine Verification

State transformation are given as a tuple of $(s_0, p, [?|!]m, s_1)$, where s_0 is the source state, p a predicate (transformation only if true), $!m1$ (sending $m1$ and waiting for an acknowledgement) / $?m1$ (waiting for $m1$ to arrive), s_1 target state.
Draw the diagram including the message flows synchronizing the machines.

Composition

Draw the possibles flows through the system until end or deadlock for given parameters.

Deadlocks

List deadlock states and final states depending on the input parameters.

Discussion Micro-Services

How to design the transition towards microservices, advantages & disadvantages.

10.7 – Naming and Coordination I

DNS

Note the request flow through an given DNS hierarchy with different strategies for each server (iterative/recursive) as tuples $s_1 \rightarrow s_2$(message).

Zookeeper I

Apache Zookeeper

Nochmal nachfragen - wie detailliert?

10.8 – Naming and Coordination II

Zookeeper II

Apache Zookeeper

Nochmal nachfragen - wie detailliert?

RW-Lock

Fill out diagram showing the effect of R/W-Lock.

10.9 – Queueing

Queueing Theory

Questions on vaious applications of different Queue modells -> Cheatsheet.

Drive-In Bank Facility