Distributed Systems Technologies 2020S

Lecture 1

Enterprise applications operate in a domain, have complex business logic, require persistent data, are physically distributed, access data concurrently, integrate with other enterprise applications and are very different from each other

Abstraction and Layering

Spaghetti code has poor isolation, no replaceability and poor maintainability

=> **Abstraction** hides the working details of a subsystem (code-level vs. system-level abstraction), **generalization** replaces multiple entities performing similar functions with single construct

Layering structures application into cohesive blocks, isolates responsibilites; interface between layers should be minimal

> "All problems in computer science can be solved by another level of indirection... except for the problem of too many levels of indirection." - Fundamental Theorem of SE

Relational model has various limitations for modern application scenarios and large-scale systems (rigid schema, only batch capabilites, limited scalability) => NoSQL as an answer to those problems

Data Management

Object-relational impedance mismatch: OO has object graphs, complex nested types & inheritance; relational model has none of those => object-orientated principles and the relational model do not match

Different patterns to abstract data sources:

1. Row Data Gateway: gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language; all details of data source access are hidden behind this interface
2. Table Data Gateway: holds all the SQL for accessing a single table or view; other code calls its methods for all interactions with the database
3. Active Record: similiar to row-data gateways but also places all domain logic in the domain object; data structure should match database => one field in the class for each column (great encapsulation and good for simpler models with few compositions; bad seperation of concerns since database and domain logic is mixed, potential for code duplication)

4. Data Mapper: layer that separates in-memory objects from database representation (clean and powerful interface, domain model can ignore database; extra layer necessary and can become quite complex)

Object-relational mapping via e.g. metadata mapping, inheritance mapping or OR frameworks (e.g. Java persistence API, Hibernate)

Inheritance mapping via 3 principles:

1. Single table inheritance: 1 table that includes all properties, possibly many null values
2. Class table inheritance: 1 table per class, only class-specific properties, possibly many joins
3. Concrete table inheritance: 1 table per class, all inherited properties, possibly many table records

ORM criticism: ORM frameworks often very complex, steep learning curve, performance issues due to naive database interactions and leaky abstraction over a relational data store

Concurrency & Transactions: abstraction, distribution and concurrent data access make it hard to maintain ACID (atomicity, consistency, isolation & durability) guarantees => Java Transaction API allows for distributed transactions and pessimistic or optimistic locking on records (via EntitiyManager)

NoSQL and beyond

CAP theorem: triangle between **consistency** (all clients always have the same view of the data), **availability** (each client can always read and write) and **partition tolerance** (system works well despite physical network partitions); pick two!

BASE semantics: basically available, soft state, eventually consistent => ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux

Data storage models e.g. relational, key/value (extremely simple schema, scales easily; querying/joins? typically few guarantees), document (flexible schema, simple data access, fairly easily scalable; less flexible queries, inefficient joins), graph (optimized for graph data; hard to grasp for SQL people)

Big data management via Hadoop DFS & MapReduce: use network of many computers to solve data management tasks; used as storage backend by distributed solutions that solve massive scalability; search & indexing services (e.g. Elasticsearch, lucene, solr) integrate nicely with Hadoop

Polyglot persistence: no one data model can satisfy all requirements of a complex enterprise application, instead, we should use the “right tool for the job” => break up database monoliths (ties in well with microservice architecture; adds additional complexity)

Lecture 2

Monoliths: difficult to scale, architecture is hard to maintain/evolve, long build-test-release cycles, too much software coupling => break up monolith in **SOA** (services communicate with each other over the network via well defined interfaces/protocols) composed of **loosely coupled elements** (updating one service does not require changing any other service) that have **bounded contexts** (services are self-contained, no knowledge of internals of other services required)

Services: logically represent a business activity, are self-contained, are a black box for its consumers and may consist of other underlying services

Remote procedure calls (RPC)

Request object represents an RPC call and holds all necessary information, response object represents the result of an RPC call

Stub hides the serialization of parameters and the network-level communication in order to present a simple invocation mechanism to the caller, skeleton is responsible for dispatching the call to the actual remote object implementation

Stubs/skeletons can be generalized via templates & generation from interface descriptions (gRPC) or via dynamic proxies (mimic interface, call invocation handlers)

Transporting messages either via HTTP (easier to debug/develop; additional transport overhead) or TCP/UDP (can be very fast; harder to develop, requires rigorous specifications)

Service-based interaction

Web services foster development of loosely-coupled distributed systems => can be used between different platforms and programming languages

Principles of SOAP/WSDL:

- Elements
    - SOAP: Simple Object Access Protocol
    - WSDL: Web Service Description Language
    - UDDI: Universal Description Discovery and Integration (discovery protocol)
    - Service consumer: User of a service
    - Service provider: Entity that implements a service (=server)
    - Service registry: Central place where avilable services are listed and advertised for lookup
- Processes
    - Publish: The service provider publishes its service with UUID carried in SOAP messages
    - Find: The service consumer looks up a suitable service using UDDI carried in SOAP- messages
    - Bind: The service consumer binds to the service provider by sending a SOAP-request.

REST is NOT a protocol, NOT a standard, NOT a particular technology => it is a resource-orientated, stateless client-server architecture with addressability and uniform interface for distributed systems, emphasizes the use of protocols and standards

Message-orientated interaction

Messaging: need for time and space decoupled integration of applications => event-driven and asynchronous messages and channels

Enterprise integration is the task of making separate applications work together to produce unified set of functionality => messaging-based integration: each applicaion connects to common messaging system where data is exchanged and behaviour is invoked using messages

Different messaging patterns e.g. point-to-point, publish/subscribe, competing consumers

Reduce number of channels by adding message-orientated middleware (MOM)

Evolution of architectural styles

Microservices have loose coupling, high cohesion and a bounded context (+ decomposes monolith, loose coupling through well defined interfaces, clear boundaries allow independent development/deployment, scale independently; - distribution adds complexity, transactions over partitioned databases are difficult, testing much harder)

Lecture 3

Coordination algorithms

Distributed consensus: how dow we make sure all nodes in a network have the same information?

Two-Phase commit: voting phase & commit phase (either commit if all nodes have commited or rollback in case voting has failed

Delegated authorization with OAuth

Direct web form authorization often insufficient for modern web applications, delegated authorization gives an app limited access to a resource that the app can access on our behalf

> The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf.

OAuth Terminology:

- Resource owner (you or me: we have data)
- Client (the application that wants the data)
- Authorization server (e.g. accounts.google.com)
- Resource server (API that holds the data, e.g. Google contacts) Access token (an ephemeral password)
- Grant type (Authorization Code Grant => assurance that the user clicked “yes”, Implicit Grant, Password Grant, Client Credentials Grant)
- Redirect URI (callback: endpoint to call during flow steps)
- Front channel (Untrusted front-facing channel, e.g. a browser)
- Back channel (Trusted back-facing channel, e.g. backend server calls)

Application frameworks

Enterprise applications often have similar and generizable functionality (e.g. persistence, transactions, security, user managament, DI) => application frameworks provide this functionality in a modularized and reuseable way, which enables code reuse, faster development and separation of concerns

Programming dynamic features

Core concerns (business logic) vs cross-cutting concerns (system-level, peripheral requirements; e.g. authentication, security, transactions)

Cross-cutting concerns may be separated through decomposition, modularization or IoC

> Metaprogramming is a programming technique in which computer programs have the ability to treat other programs as their data. It means that a program can be designed to read, generate, analyze or transform other programs, and even modify itself while running.

Metaprogramming overview

- Reflection (allows a program to inspect itself, analyze program structures at runtime, runtime invocation of methods; programs become more difficult, static type checking is lost => higher risk of errors)
  - OO meta model
  - Java reflection mechanisms Annotations
  - Dynamic proxies
- Instrumentation
  - Bytecode injection
  - Javassist, Bytebuddy

AOP

Main Concepts of AOP

- Aspect: modularized implementation of a cross-cutting concern
- Join Point: an identifiable point in the execution of a program (e.g., method invocation, field access, constructor call, ...)
- Pointcut: an expression that selects join points and collects context about those points (e.g., target object and arguments of a method invocation)
- Advice: the code to be executed at a join point, can execute before, after, or around the join poin

Weaving: process of merging aspects into the program code, 4 methods:

- Compile-time weaving: compile from sources and produce woven class files
- Post-compile/binary weaving: weave aspects into existing class files or JARs
- Load-time weaving: deferred binary weaving (weave when class is loaded)
- Run-time weaving: weave at runtime via dynamic proxies (NOT supported by AspectJ)

Lecture 4

Virtualization and isolation

> Virtualization refers to the act of creating a virtual (rather than actual) version of something, including virtual computer hardware platforms, storage devices, and computer network resources.

Hardware virtualization (VMs) vs OS-level virtualization (containers)

Virtualization software runs on a physical server (the host) to abstract the host's underlying physical hardware, the underlying physical hardware is made accessible by the virtualization software (or Hypervisor, or Virtual Machine Monitor)

Type 1 (native) takes the place of the OS, thus removes the OS layer => common in data centers; type 2 (hosted) hypervisor is installed in host's running OS, therefore great for desktop and consumer products

OS-level virtualization

Kernel space: on modern systems with protected memory management units, the kernel typically resides in an elevated system state, which includes a protected memory space and full access to the hardware. This system state and memory space is collectively referred to as kernel-space.

User space: applications execute in user-space, where they can access a subset of the machine's available resources and can perform certain system functions, directly access hardware, access memory outside of that allotted them by the kernel, or otherwise misbehave.

=> When executing kernel code, the system is in kernel-space executing in kernel mode. When running a regular process, the system is in user-space executing in user mode. Applications (running in user-space)

communicate with the kernel via system calls

> A container is a self contained execution environment that shares the kernel of the host system and which is (optionally) isolated from other containers in the system.

OS-level virtualization refers to an operating system paradigm in which the kernel allows the existence of multiple isolated user-space instances, which may look like real computers from the point of view of programs running in them

Docker is a set of tools around container management (API/CLI, volume/network manager, container image build tool)

VMs vs containers

VM-based deployment: VMs as unit of isolation; each VM includes fully-fledged guest OS, binaries, libraries and application; great isolation; fat; flexible

Container deployment: containers as unit of isolation; containers include binaries, libraries and application; runs in user-space and thus shares host OS kernel; less isolation; lightweight; less flexible since it cannot run in arbitrary guest OS

Performance: some CPU intensive algorithms (like Linpack) can use OS/hardware optimization that is lost when using VMs, but maintained using Docker (because it shares the Host's Kernel); for memory or sequential IO, the performance is almost the same

> CoreOS – Container Linux: lightweight OS for hosting containers. No package manager, only containers!
>
> Kubernetes: scalable Container Orchestration Platform

Container orchestrating

Problem: cluster management, containers as deployment unit, container engines usually only serve one node

=> Solution: container orchestration: orchestrate multiple container engine nodes, automated applicaton deployment and operations, manage cluster resources, scalability

Fundamental operational mechanisms: resource management, scheduling, logging, autoscaling, load balancing

Autoscaling

> Elastic computing is the use of computer resources which dynamically to meet a variable workload

> Clustering elasticity is the ease of adding or removing nodes from the distributed data store

Types of Auto Scaling

- Reactive: monitor performance metrics and adjust capacity to maintain performance
- Proactive: leverage known or previously observed load patterns (e.g., increased load during the day)
- Predictive: predict load based on pattern recognition from historical data, and scale out accordingly

Load balancing strategies e.g. round robin, weighted round robin, hash-based, least requested/utilized/response time

Lecture 5

Event-based systems (EBS)

> An event-based system is a system in which the integrated components communicate by generating and receiving event notifications.

An event is an occurence of a happening of interest, a notification describes an event (what happened when, what are the changes) => we commonly use "event" for event notification

EBS comprises (among others, terminology is messy):

- Event-Driven Architecture
- Event Stream Processing
- Complex Event Processing

Event-driven architecture

UIs are naturally event-driven

Architectural patterns e.g. event notification, event-carried state transfer, event sourcing

Event notification: request-response may take long => emit event object to e.g. event queue (encapsulates what happenend, reverses dependency and decouples sender from receiver)

Event-carried state transfer: events carry the state of the change in their body (allows components to store data relevant to them, more decoupling, removes dependencies; more event traffic and more storage required)

Event sourcing: usually changing records involve removing old and adding new record, event sourced systems track each change as event and these events are applied to application state to change aspects of the state

EBS are often built on top of messaging systems since they can distribute events via messages, route,

filter, enrich & aggregate

(Event) data streams: application events, user profile changes, clicks, ...

Stream processing used to create topology of these streams (implementations e.g. Apache Storm, Apex, Samza, Flink)

Event time (when event ocured) vs processing time (when system got the event)

Monitoring & log analysis

Monitoring architecture: instrumentation, collection, storage, analytics, notification, visualization

Instrumentation: black box (CPU utliization, RAM, disk space, network traffic) vs white box (HTTP status codes, logged in users, queries/second, total clicks)