Distributed Architecture, Interaction, and Data Models

Hong-Linh Truong
Faculty of Informatics, TU Wien

hong-linh.truong@tuwien.ac.at
http://www.infosys.tuwien.ac.at/staff/truong
Twitter: @linhsolar
Ack:

Some slides are based on previous lectures in SS 2013-2015
Outline

- Overview
- Key design concepts
- Architecture styles and Interaction Models
- Data models
- Optimizing interactions
- Summary
DST Lectures versus Labs

- Cover some important topics in the current state-of-the-art of distributed systems technologies
  - We have focusing topics
- Few important parts of the techniques for your labs
  - Most techniques you will learn by yourself
- Stay in the concepts: no specific implementation or programming languages
DST Lectures versus Labs

- It is **not** about Java or Enterprise Java Beans!
  - The technologies you learn in the lectures are for different applications/systems.
Have some programming questions?

Join the Stack Overflow community to:

- Ask programming questions
- Answer and help your peers
- Get recognized for your expertise

Stack Overflow is a community of 4.7 million programmers, just like you, helping each other.

Join them; it only takes a minute:

Sign up

Or send the questions to the tutors
Where is our focus?

Backend versus front-end
Figure source - https://www.upwork.com/hiring/development/a-beginners-guide-to-back-end-development/

Full stack developer
Figure source - https://medium.com/dev-bits/why-full-stack-development-is-too-good-for-you-in-2017-3fd6fe207b34

DST topics:
Backend services in multi-cloud environments

DST topics:
Communications with Front-end

Non DST topics
Front-end
TRENDS & KEY DESIGN CONCEPTS
Rapid changes in application requirements and technologies for distributed applications

- On-premise servers → public clouds and on-premise clouds
- Static, small infrastructures → large-scale virtualized dynamic infrastructures
- Heavy monolithic services → microservices
- Server → Serverless Architecture
- Data → Data, Data and Data
A not so complex distributed application

Technologies

Distribution

Figure source: https://docs.oracle.com/javaee/7/tutorial/overview003.htm

Figure source: http://drbacchus.com/files/serverrack.jpg
A complex, large-scale distributed system
What we have to do?

System/application business logic

- Data
- Communication
- Processing
- Visualization
- Routing
- Load balancing
- Monitoring & Logging
- Etc.

Development and operation tasks

- Development
- Deployment
- Testing
- Monitoring
- Performance analysis
- Teamwork

selecting the right technologies as well as design methodologies
Understand the requirements

- Data
  - Structured, semi-structured or unstructured data?
  - Do we need data being persistent for several years?
  - Is accessed concurrently (from different applications)?
  - Mostly read or write operations?

- Data intensive or computation intensive application

This course is not about big data but distributed applications today have to handle various types of data at rest and in motion!
Understand the requirements

- Physically distributed systems
  - Different clients and back-ends
  - On-premise enterprise or cloud systems?
- Complex business logics
  - Complexity comes from the domain more than from e.g., the algorithms
- Integration with existing systems
  - E.g., need to interface with legacy systems or other applications
- Scalability and performance limitation
- Etc.
How do we build distributed applications

- Using fundamental concepts and technologies
  - Abstraction: make complicated things simple
  - Layering, Orchestration, and Chorography: put things together
  - Distribution: where and how to deploy
- Using best practice design and performance patterns
- Principles, e.g., Microservices Approach

Figure source: Sam Newman, Building Microservices, 2015
Abstraction

Deal with technical complexity by hiding it behind clear simple interfaces

- APIs abstracting complex communications and interactions
- Interfaces abstracting complex functions implementation
Deal with maintainability by logically structuring applications into functionally cohesive blocks

Benefits of Layering

- You can understand a single layer without knowing much about other layers
- Layers can be substituted with different implementations
- Minimized dependencies between layers
- Layers can be reused

Downsides of Layering

- Layers don’t encapsulate all things well: do not cope with changes well.
- Extra layers can create performance overhead
- Extra layers require additional development effort
Examples: abstraction and layering side-by-side

Figure source: http://docs.jboss.org/hibernate/orm/5.1/userguide/html_single/Hibernate_User_Guide.html
Partitioning functionality & data

- Why?
  - Breakdown the complexity
  - Easy to implement, replace, and compose
  - Deal with performance, scalability, security, etc.
  - Support teams in DevOps
  - Cope with technology changes

Enable abstraction and layering/orchestration, and distribution
Example of functional and data partitioning

Figures source: http://queue.acm.org/detail.cfm?id=1971597
Partitioning functionality: 3-Layered Architecture

- **Presentation**
  - Interaction between user and software

- **Domain Logic (Business Logic)**
  - Logic that is the real point of the system
  - Performs calculations based on input and stored data
  - Validation of data, e.g., received from presentation

- **Data Source**
  - Communication with other systems, usually mainly databases, but also messaging systems, transaction managers, other applications, ...
Orchestration and Choreography

Orchestration

Sensor Data Analytics

Energy Optimization Service

Emergency Service

Equipment Maintenance Service

Sensors

Near Realtime Analysis

Historical Data Archiving

Choreography
Distribution: where to run the layers?

Figure source: Andrew S. Tanenbaum and Maarten van Steen, Distributed Systems – Principles and Paradigms, 2nd Edition, 2007, Prentice-Hall
Distribution: OS, VM, Container, or Function-as-a-Service?

Source: The XEN Hypervisor (http://www.xen.org/)

Docker
The Docker Engine container comprises just the application and its dependencies. It runs as an isolated process in userspace on the host operating system, sharing the kernel with other containers. Thus, it enjoys the resource isolation and allocation benefits of VMs but is much more portable and efficient.
Distribution: edge systems, core network backbone or data centers?

Use Case 3: Video Analytics

Figure 4: Example of video analytics
Figure source: https://portal.etsi.org/portals/0/tbpages/mec/docs/mobile-edge_computing_-_introductory_technical_white_paper_v1%2018-09-14.pdf

Chinese police are using smart glasses to identify potential suspects

Figure source: https://techcrunch.com/2018/02/08/chinese-police-are-getting-smart-glasses/
Programming

<table>
<thead>
<tr>
<th>Language Rank</th>
<th>Types</th>
<th>Spectrum Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Python</td>
<td>🌍️📱💻</td>
<td>100.0</td>
</tr>
<tr>
<td>2. C</td>
<td>💻📱💻</td>
<td>99.7</td>
</tr>
<tr>
<td>3. Java</td>
<td>🌍️📱💻</td>
<td>99.5</td>
</tr>
<tr>
<td>4. C++</td>
<td>🌍️📱💻</td>
<td>97.1</td>
</tr>
<tr>
<td>5. C#</td>
<td>🌍️📱💻</td>
<td>87.7</td>
</tr>
<tr>
<td>6. R</td>
<td>🌍️📱💻</td>
<td>87.7</td>
</tr>
<tr>
<td>7. JavaScript</td>
<td>🌍️📱💻</td>
<td>85.6</td>
</tr>
<tr>
<td>8. PHP</td>
<td>🌍️📱💻</td>
<td>81.2</td>
</tr>
<tr>
<td>9. Go</td>
<td>🌍️📱💻</td>
<td>75.1</td>
</tr>
<tr>
<td>10. Swift</td>
<td>🌍️📱💻</td>
<td>73.7</td>
</tr>
</tbody>
</table>

Source: https://spectrum.ieee.org/computing/software/the-2017-top-programming-languages
What is the downside of functional and data partitioning?
ARCHITECTURE STYLES AND INTERACTION MODELS
Using abstraction, we hide the complexity within these boxes.

But we need to integrate between two components, enabling them communicate across process boundaries.

- In the same host, in the same application in different hosts, in different applications.
- How would they exchange data/commands? e.g., Synchronous or asynchronous communication.

Complex in context of complex distributed systems.
Basic interaction models

- Large number of communication protocols and interfaces
- Interaction styles, protocols and interfaces
  - REST, SOAP, RPC, Message Passing, Stream-oriented Communication, Distributed Object models, Component-based Models
  - Your own protocols
- Other criteria
  - Architectural constraints
  - Scalability, performance, adaptability, monitoring, logging, etc.
Component Based Systems

- Components:
  - Reusable collections of objects
  - Clearly defined interfaces
  - Focus on reuse and integration

- Implementations: Enterprise Java Beans, OSGi, System.ComponentModel in .NET
Service-Oriented Systems

- Service-oriented Computing:
  - Applications are built by composing (sticking together) services (lego principle)
- Services are supposed to be:
  - Standardized,
  - Replaceable,
  - Reusable/Composable,
  - Stateless
## Components vs. Services

<table>
<thead>
<tr>
<th>Components</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Tight coupling</td>
<td>- Loose coupling</td>
</tr>
<tr>
<td>- Client requires library</td>
<td>- Message exchanges</td>
</tr>
<tr>
<td>- Client / Server</td>
<td>- Policy</td>
</tr>
<tr>
<td>- Extendable</td>
<td>- Peer-to-peer</td>
</tr>
<tr>
<td>- Fast</td>
<td>- Composable</td>
</tr>
<tr>
<td>- Small to medium granularity</td>
<td>- Some overhead</td>
</tr>
<tr>
<td>- Buying components and installing them on</td>
<td>- Medium to coarse granularity</td>
</tr>
<tr>
<td>your HW</td>
<td>- Pay-per-use on-demand services</td>
</tr>
</tbody>
</table>

**Components:**
- Tight coupling
- Client requires library
- Client / Server
- Extendable
- Fast
- Small to medium granularity
- Buying components and installing them on your HW

**Services:**
- Loose coupling
- Message exchanges
- Policy
- Peer-to-peer
- Composable
- Some overhead
- Medium to coarse granularity
- Pay-per-use on-demand services
REST

- **REST**: REpresentational State Transfer
- Is an architectural style! (not an implementation or specification)
  - See Richardson Maturity Model (http://martinfowler.com/articles/richardsonMaturityModel.html)
  - Can be implemented using standards (e.g., HTTP, URI, JSON, XML)
- **Architectural Constraints:**
  - Client-Server, Stateless, Cacheable, Layered System, Uniform Interface
Example of REST Interactions

- Important concepts
  - Resources
  - Identification of Resources
  - Manipulation of resources through their representation
  - Self-descriptive messages
  - Hypermedia as the engine of application state (aka. HATEOAS)

Diagram:
- Web Service Client
  - GET (list/retrieve)
  - PUT (update/create)
  - POST (create/update)
  - DELETE (remove)
- Web Service
  - URI: Resource
  - URI: Resource
Recall: Remote Procedure Call Systems

- Server provides procedures that clients can call
- Most RPC-style middleware follows a small set of architectural principles
- Strongly tied to specific platforms
- Why is it relevant in complex distributed systems?
What kind of benefits we get, compared with REST Interactions and data exchange formats?
Server-sent Events and WebSocket

- Server-sent Events
  - Remember polling results from servers?
  - Server pushes data to clients through HTTP when the clients connect to the server.

  - Remember socket?
  - Two ways of communication through TCP
  - Example, socket.io (more than just a typical WebSocket)

For which use cases/scenarios we can use them?
WebHook

URL accepting HTTP POST

```ruby
// Set your secret key: remember to change this to your live secret key in production
// See your keys here: https://dashboard.stripe.com/account/apikeys
var stripe = require("stripe")("sk_test_BQokikJOvBiI2HlWgH4olfQ2");

// This example uses Express to receive webhooks
const app = require("express")();

// Retrieve the raw body as a buffer and match all content types
app.use(require("body-parser").raw({type: "/\*\/\*"}));

app.post("/my/webhook/url", function(request, response) {
  // Retrieve the request's body and parse it as JSON
  var event_json = JSON.parse(request.body);

  // Do something with event_json
  response.send(200);
});
```

Source: https://stripe.com/docs/webhooks
Message Passing/Message-Oriented Communications

More in lecture 2 (fundamental) and lecture 5 (large-scale)

- Servers and clients communicate by exchanging messages

Stream-oriented communication
When delivery times matter!

Servers and clients communicate by exchanging messages.
Complex interactions

- One-to-many, Many-to-one, Many-to-One
  - Message Passing Interface
  - Public/Subscribe, Message-oriented Middleware
  - Shared Repository
  - Websocket (also support broadcast)
  - Application/Systems specific models

![Diagram of client-server interactions with a repository and various operations like add, retrieve, delete, listen, notify, and Amazon S3 with RabbitMQ and CloudMQTT logos.]
Serverless

- Most of the time we need to build and setup various services/server
- But with the cloud and PaaS providers → we do not have to do this
- Serverless computing:
  - Function as a service
- Examples
  - AWS Lambda
  - Google Cloud Function (beta - https://cloud.google.com/functions/)
  - IBM OpenWhisk
  - https://serverless.com/
Key principles

- Running code without your own back-end server/application server systems
- Tasks in your application: described as functions
  - With a lifecycle
- Functions are uploaded to FaaS and will be executed based on different triggers (e.g., direct call or events)

Event-driven triggers!

Check: https://martinfowler.com/articles/serverless.html
Example: chat

Case study serverless Deloitte-Amtrak

Source: https://www.slideshare.net/GaryArora/leapfrog-into-serverless-a-deloitteamtrak-case-study-serverless-conference-2017/

Value Delivered
Developed and released in six months!

- Processing 1 million transactions/day with a peak load of 2K transactions/minute
- near real-time reports and dashboards
- Single source of truth & entry via JSON restful services
- NoSQL schema: Future proof
- Improved data accuracy by supporting edge cases that were previously missed
- Laid out the groundwork for decommissioning legacy systems
- Low cost maintenance & operations: No servers to maintain, load-balance, or scale

Leapfrog into Serverless - a Deloitte-Amtrak Case Study | Serverless Conference 2017
Depending on the requirements: we can build everything or build few things and manage the whole system or not.

→ We need to carefully study and examine suitable technologies/architectures for our complex distributed applications

A big homework:
Microservices approach versus serverless approach
DATA MODELS
Data Storage Models

- Relational Model
  - Traditional SQL model
- Key-Value Model
  - Data is stored as simple list of keys and values (hashtable style)
- Column-oriented Model
  - Data is stored in tables, but stored column-wise rather than row-wise
- Document-oriented Model
  - Data is stored in (schemaless) documents
- Graph-oriented Model
  - Data is stored as an interconnected graph
Relational Model

- Implemented as collection of two-dimensional tables with rows and columns
- Powerful querying & strong consistency support
- Strict schema requirements
- E.g.: Oracle Database, MySQL Server, PostgreSQL
Key-Value Model

- Basically an implementation of a map in a programming language
- Values do not need to have the same structure (there is no schema associated with values)
- Primary use case: caching
- Simple and very efficient, fast (e.g., in memory storage)
- Querying capabilities usually very limited
  Oftentimes only “By Id” pattern
- E.g.:
  - Memcached, Riak, Redis
Document-oriented Model

A simple analogy

- Simple, comparable to key-value
- All values are schema-free and typically complex
- Primary use cases: managing large amounts of unstructured or semi-structured data
- Sharding and distributed storage is usually well-supported
- Schema-freeness means that querying is often difficult and/or inefficient
- E.g.: CouchDB, MongoDB
Example: MongoDB with mLab.org
Complex Relationships?

How do we represent such relationships with documents?
Column-oriented data model

- Data Model
  - Table consists of rows
  - Row consists of a key and one or more columns
  - Columns are grouped into column families
  - A column family: a set of columns and their values

- Systems: Hbase, Hypertable, Cassandra

Rows are allowed to have different columns
Examples: HBase

<table>
<thead>
<tr>
<th>Row Key</th>
<th>Time Stamp</th>
<th>ColumnFamily contents</th>
<th>ColumnFamily anchor</th>
<th>ColumnFamily people</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t9</td>
<td></td>
<td>anchor: cnnsi.com = &quot;CNN&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t8</td>
<td></td>
<td>anchor: mylook.ca = &quot;CNN.com&quot;</td>
<td></td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t6</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t5</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;com.cnn.www&quot;</td>
<td>t3</td>
<td>contents:html = &quot;&lt;html&gt;...&quot;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source: http://hbase.apache.org/book.html#datamodel
Graph-oriented Model

- Data relationships as first-class citizens
- Data is stored as a network (graph)
- Primary use cases: whenever one is more interested in the relations between data than the data itself (for instance, social media analysis)
  - Highly connected and self-referential data is easier to map to a graph database than to the relational model
  - Relationship queries can be executed fast
- E.g.: Neo4J, Orient DB, ArangoDB
  - Many of them are actually multi-model (combine graph, document, key/value, etc., models)
Blockchain as a database

Meet BigchainDB. The blockchain database.

Get Started

Bigchain DB: https://www.bigchaindb.com/

Key issues: we need to use many types of databases/data models

Example - Healthcare

- Personal or hospital context
- Very different types of data for healthcare
  - Electronic Health Records (EHRs)
  - Remote patient monitoring data (connected care/telemedicine)
  - Personal health-related activities data
- Combined with other types of data for insurance business models
If you have to build a system that includes many individuals connected through a SocialNetwork for discussing products they sell and buy and they have a lot of different products to sell

How would you select database technologies for your implementation?
Accessing and Processing Data

- Component accesses data
  - Get, store, and process
  - Data is in relational model, documents, graph, etc.

- Main problems
  - Programming languages are different → Mapping data into objects in programming languages
  - Distributed and scalable processing of data (not in the focus of this lecture)
Data Access API Approach

- Data access APIs can be built based on well-defined interfaces
- Currently mostly based on REST
- Help to bring the data objects close to the programming language objects
SQL-based API

- Leverage SQL as the language for accessing data
  - Hide the underlying specific technologies

Source: Programming Hive, Edward Capriolo, Dean Wampler, and Jason Rutherglen
Conceptual mismatch, especially with relational database

Programming Language Objects

Native Database Structure (e.g., relations)
What you want to avoid

Not just about security but tedious effort on coding

String query = "SELECT account_balance FROM user_data WHERE user_name = " + request.getParameter("customerName");

try {
    Statement statement = connection.createStatement( ... );
    ResultSet results = statement.executeQuery( query );
}

Source: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
Solution (1)

Build an abstraction layer that represents the database in the application

Two subproblems:
1. How do represent data in the application?
2. How to map between data storage and application?
Solution (2)

- Technologies
  - Java Persistence API
  - Hibernate ORM (relational database)
  - Hibernate OGM (NoSQL)
  - Mongoose (for MongoDB)

- Methodology: design patterns
**Data-Related Architectural Patterns**

- See [http://martinfowler.com/eaaCatalog/index.html](http://martinfowler.com/eaaCatalog/index.html)

- Mapping DB Data to Code
  - Code that wraps the actual communication between business logics and data store
  - Required to „fill“ e.g., the domain model

- Goals
  - Access data using mechanisms that fit in with the application development language
  - Separate data store access from domain logic and place it in separate classes
Data Source Architectural Patterns

- **Row Data Gateway**
  - Based on table structure. One instance per row returned by a query.

- **Table Data Gateway**
  - Based on table structure. One instance per table.

- **Active Record**
  - Wraps a database row, encapsulates database access code, and adds business logic to that data.

- **Data Mapper**
  - Handles loading and storing between database and Domain Model
Object-Relational Structural Patterns

Association Table Mapping

Class Table Inheritance

Solutions/Strategies:
http://docs.oracle.com/javae6/tutorial/doc/bnbqn.html#bnbqr
Do the loading as latest as possible

Source: https://martinfowler.com/eaaCatalog/lazyLoad.html
Lazy Loading

- For loading an object from a database it's handy to also load the objects that are related to it
  - Developer does not have to explicitly load all objects

Problem

- Loading one object can have the effect of loading a huge number of related objects

- Lazy loading interrupts loading process and loads data transparently when needed
Lazy Loading Implementation Patterns

- Lazy Initialization
  - Every access to the field checks first to see if it's null

- Value Holder
  - Lazy-loaded objects are wrapped by a specific value holder object

- Virtual Proxy
  - An object that looks like the real value, but which loads the data only when requested

- Ghost
  - Real object, but in partial state
  - Remaining data loaded on first access
Lazy Loading Example - Hibernate

@javax.persistence.Entity
public class Product {
    @javax.persistence.OneToMany(mappedBy="product", fetch = FetchType.LAZY)
    // or FetchType.EAGER for eager loading
    public Set<Contract> getContracts() {
        ...
    }
}

How can we achieve the implementation? using proxy technique (Lesson 3)
OPTIMIZING INTERACTIONS
Interactions?
Optimizing Interactions

- Interactions between software components and within them
- Scale in: increasing server capability
- Load balancer
- Scale out
- Asynchronous communication
  - More in lectures 4&5
- Data sharding
- Connection Pools
- Etc.
Scale out

Figure source: http://queue.acm.org/detail.cfm?id=2560948

More in Lecture 4
Load balancing

Figure source: http://queue.acm.org/detail.cfm?id=1971597
Need also Routing, Metadata Service, etc.

Source: https://docs.mongodb.org/manual/core/sharding-introduction/
DST 2018

Prevent too many accesses?

Client → 100000 requests/s → Service

Client → API Management Service → Service

Code: http://www.django-rest-framework.org/api-guide/throttling/#how-throttling-is-determined
Other patterns

- See the supplement slides
- Understanding how to use communications to implement certain patterns
  - Polling
  - Fire and forget
  - Callback
Summary

- Understand the size and complexity of your distributed applications/systems
- Pickup the right approach based on requirements and best practices
- Architecture, interaction, and data models are strongly inter-dependent
- There are a lot of useful design patterns
- Distribution design and deployment techniques are crucial → cloud models
- Embrace diversity: Not just distributed applications with relational database!
Other references

- Sam Newman, Building Microservices, 2015
- http://de.slideshare.net/spnewman/principles-of-microservices-ndc-2014
- Markus Völter, Michael Kirchner, Uwe Zdun: Remoting Patterns – Foundation of Enterprise, Internet and Realtime Distributed Object Middleware, Wiley Series in Software Design Patterns, 2004
- Roy Fielding’s blog entry on REST requirements: http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven
- Martin Fowler’s blog entry on RMM: http://martinfowler.com/articles/richardsonMaturityModel.html
- Martin Fowler: Patterns of Enterprise Application Architecture
- Eric Redmond, Jim R. Wilson: Seven Databases in Seven Weeks – A Guide to Modern Databases and the NoSQL Movement
- Polyglott persistence: http://martinfowler.com/bliki/PolyglotPersistence.html
- Eventual consistency: http://queue.acm.org/detail.cfm?id=1466448
- https://hackernoon.com/blockchains-versus-traditional-databases-c1a728159f79
- https://www.oracle.com/cloud/blockchain/index.html#compare
Thanks for your attention

Hong-Linh Truong
Faculty of Informatics, TU Wien
hong-linh.truong@tuwien.ac.at
http://www.infosys.tuwien.ac.at/staff/truong