

CS 331 Notes

Table Of Contents

- Lecture 1 - page 2: Key topics we will learn; Scale; Factors to consider.
 - page 3: SDLC; Domains of LSSD; Middleware.
- Lecture 2 - page 4: Distributed Systems; Application, Transport and Network OSI
 - page 5: TCP; Sockets; Java Serialization
- Lecture 3 - page 6: Types of Middleware: Socket, RMI, SEO; HTTP Protocol
 - page 7: Servlets; Software Lifecycles; REST
 - page 8: REST; HTTP Methods / codes
- Lecture 4 - page 9:

Lecture 1

Key Topics of 331:

- Issues that arise in LLSD
- Teamwork
- Distributed Systems
- Web Services
- Representational State Transfer (REST)
- Resource Representation
- Data Management
- ORM
- Transactions
- Middleware
- Software Architecture
- Architecture Tactics and Patterns

Lecture 1 Topics:

- What is Scale
- Factors that affect the scale of LSSD
- Life Cycles of SD
- Architecture (middleware)

What is Scale:

Scale is a multi-factored consideration when building software. It affects the team structure and composition alongside the design process and life cycle.

It is partially described by the project costs, required components and understanding of the problem description.

It is not a simple enough problem to be simply described by the cost of the program or a few simple measures.

Factors that affect scale / risks of LSSD:

Scope - Consider how much of the problem needs to be mapped correctly to the program, and reduce the scope for anything unnecessary.

Technology - Consider the program to decide which stack is necessary to build an effective solution (consider the language (static / dynamic typing, etc.) and consider what the stack is optimised for.

Process - Consider the workflow process and how to split the program into subprograms that different groups / teammembers can work on separately.

Resources - Consider the amount of resources to allocate and whether reallocation of resources may be necessary when building teams. Plan for these potential changes. Write code so that replacement can easily interpret and work with built code.

Security and Compliance - Ensure dangerous tasks are taken care of safely. Don't save credit card details and hash passwords. Use type safety to reduce runtimes.

Performance - Consider where performance gets the most effective remember premature optimisation is the root of all evil.

Consider continuous integration and deployment - auto track performance measures.

Budget - Consider how much budget exists, don't build over the top and waste more than you have.

CS331 Notes

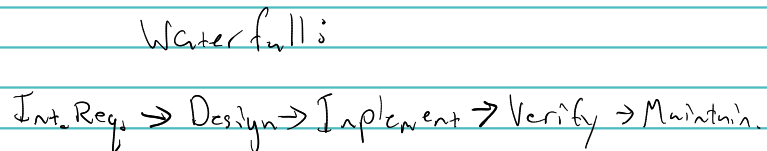
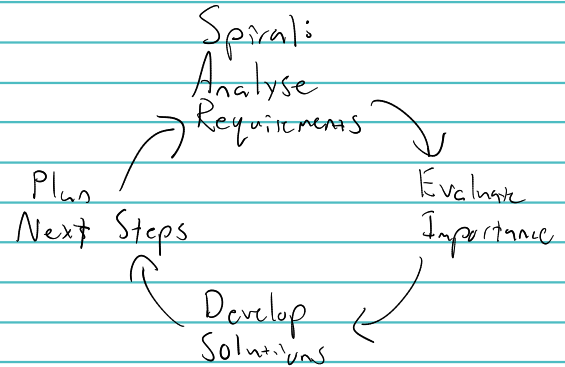
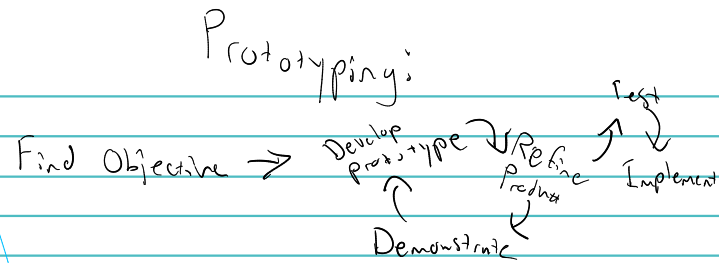
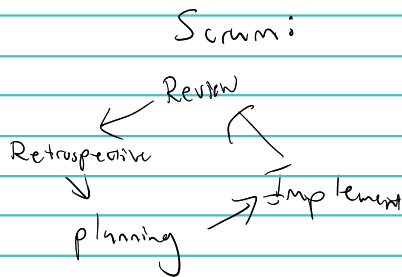
Software Development Life Cycle:

There are different Life Cycles for SD.

These include prototyping, spiral, waterfall and agile.

In class we consider Agile the best because it scales effectively. However for smaller projects it may not be as effective as there are a lot of unnecessary procedures.

Agile Attempts to solve team-work issues. To do so, they consider scrum masters who know how to communicate with the developers and explain this to the project management team. (loses direct communication, so can be very harmful).



Domains of LSSD include:

- Enterprise Resources / Process Management (finance, supply chain etc.)
- Digital Customer Channels (Ecommerce Advertising etc.)
- Industrial Software (Equipment monitoring, Quality management, Analytics etc.)
- Smart connected products (Remote monitoring, Smart devices etc.)
- Learning networks (Digital Workspaces, eLearning etc.)
- Emergency Services (ERS management, Info Sys, Security, Surveillance etc.)

These often require Distributed Services (pg fill me in)

Client middle and service layers contrasting to standalone products.

What is middleware?

Middleware is software that allows different machines to communicate with each other easily. It adds abstractions that allow the user to use simple functions to solve problems that may be more complex / change if we were to directly speak to the other servers. For example the conversion between Javascript and Java that has to happen whenever web servers communicate.

Often communicates this data using common protocols like SOAP, REST or JSON.

We want middleware because it allows us to have a standard package to ensure:

- Consistent Safety (not necessarily safe, but everywhere is equally safe).
- Reuse of Code leading to easy global modification if tasks are unsafe.
- less complex DEVXP because of simple functions to handle complex transactions.

Lecture 2 - Distributed Systems

Topics:

- What are Distributed Systems
- Network Infrastructure
 - Net Protocols, Internet Protocols, TCP TCP with Java
- Java Serialization

A distributed System is a system with multiple components located on different machines that are used to coordinate actions as a single coherent system to the end-user.

- This may include a desktop client, database server, cloud service etc.

Major problems in distributed systems include:

- Exactly-one delivery (where two systems send the same data, causing duplicate data).
- guaranteed order of messages (where data is sent from two systems at different times causing incorrectly ordered results).

Network Infrastructure problems:

- Sometimes computers and links fail independantly.
- Switches have finite space per packet.
- Links vary in terms of capacity.
- Data can be corrupted.
- Incorrect delivery of packets.

Network Protocols:

We organise the protocols into layers that abstracts different components of the network. The application layer handles the interaction between clients and servers. It ensures that clients send correct types of information. It also defines what kind of information that can be expected. Finally it defines rules for when and how to send these messages.

Major use cases: data integrity, timing, guaranteeing throughput and security (though this can be handled on the transport layer.)

Transport Layer:

TCP provides reliable transport, flow control, congestion control and is connection-oriented. It does not provide anything stated in the major use cases for network protocols.

UDP provides unreliable data transfer or any of the other uses. However this can be extended on the application layer to allow more custom control for more unusual programs

Network Layer provides a packet delivery service between different host machines.

This allows different machines to communicate and ensures the data is sent to the correct device.

The layers depend on the interface of the lower levels, not the implementation.

An IP is the way the network layer distinguishes between different host devices. We send a packet from one device to another sending the source IP address, destination IP address and a payload of the data we want to send to the destination.

TCP (transmission control protocol) is a transport protocol that creates a virtual connection between a pair of processes.

It allows the application layer above it the ability to send data without worrying about:

- the limited message sizes of a packet.
- the destination of the data (i.e. ports and IP addresses)
- the chance of data being lost / corrupted
- message duplication
- flow control

One way it could do this is with a log, timer, by sending acknowledgement packets with each packet, a sequence number, a checksum and a buffer.

To handle message sizes, it can keep a buffer of all sending information and break it down into payload sizes and send them one at a time. To handle corrupt data, it can send use a checksum that is invalid if data was corrupted, and the other recipient can send back a false acknowledgement packet to tell the host machine to resend the data.

To handle lost data, it can resend the packet if it takes to long.

To handle flow control, it can use the sequence number to ensure that even if the data is recieved in the wrong order, the recipient can check the sequence number and wait for any missing data.

A socket is bound to an IP address and a port number and it identifies where the application should send data to, and recieve data from.

A socket converts the data to be sent to be something the internet can interpret.

To allow this to happen, we need to describe each class as serializable, as this forces the compiler to decide on how to serialize it into a primitive datatype.

The java serializer describes what an object looks like, this allows the client to rebuild each object by deserializing it.

When an object is serialized, it becomes a tuple of features consisting of: the class name, it's version number, it's handle, the number of instance variables it has, the name of each variable and the type of each variable.

Then we also send a set of objects that consist of the list of the class handle, followed by each instance variable for each class.

Serializing data allows us to send data to the network easily, save it to a file and make a deep copy in memory. Transient data will not be sent.

Lecture 3 - Web Services

Topics:

- What kinds of middleware are there?
- Service-oriented architecture (SOA)
- HTTP
- Servlets / servlet containers
- Web Services and REST

Types of Middleware:

Socket Programming Middleware:

Socket programming based middleware is a type of value-driven middleware:

It only abstracts away the TCP component, so is very low-level as the developer has to work with streams manually and pass the values directly to through the stream.

Remote Method Invocation Middleware:

This has abstracted away the socket, and instead the client communicates with a proxy for the server. This allows the programmer to avoid directly handling streams and communication occurs through invoking object proxies and methods.

This works by having an interface that defines how the client can interact with the server and an interface for how the server can interact with the server. Then it calls an implementation of that interface that handles the socket-programming (value-driven.)

Service-Oriented Middleware:

This uses a series of services that communicate with a defined format that is not necessarily directly passing the objects. This allows for the client and server to communicate in different languages.

For example, the server could work in Java, but the client might only understand javascript, to work around this, they could communicate using JSON, then interpret what the other says into a more useful format.

To do this, we need to define a serialization that allows us to marshal / demarshal the problem into JSON and back from JSON respectively.

We communicate by invoking requests in JSON or XML because we cannot directly call methods as we don't know the language we are calling to / from.

It is very useful when a lot of services have vastly different expectations (programming languages, data types, information about data etc.)

however it's not that effective for performance or cost (because of loss of potential gains from not needing some part of the abstraction sometimes [such as services using the same language].)

Look further into architectures with: <https://www.developertoarchitect.com/lessons/>

HTTP Protocol:

uses two kinds of messages: requests and responses

A request is constructed with the method, the url, the version, a header and a body.

A reply is a version, a status, a reason, a header and a body.

A url has `http://serverName[:port] [/pathName] [?query]`.

Servlets:

Servlets are middleware that abstract away the http requests and sessions. They control how the program works based on the model and the view.

Servlets are singletons that handle each server. If a servlet gets multiple threads of requests, it needs to handle them one at a time for thread safety.

A servlet container manages the different servlets and the lifecycle of the servlets.

Lifecycle:

the servlet is first initialised, then performs its serve for each request it's given using the `serve()` method. When the servlet container is shutting down / resources need to be freed, the servlet is destroyed and the `destroy()` method is ran which frees the data the servlet is holding and backs up any potentially dangerous problems.

The `HTTPServlet` class is an abstract class that has `doGet`, `doDelete`, `doPost` and `doPut` methods that handle the different kinds of interactions a user will have with the server. To make our own servlet, we will override these methods to allow users to get results.

These methods take in a Request and Response object (we do not call these methods directly, so we don't need to worry about that.) but these allow us to write using the `resp.getWriter()` method.

When doing this, we also need to make a `web.xml` file that tells the program the structure of the file.

REST:

REST stands for Representational State Transfer, was designed by Roy Fielding.

There are 5 main principals of REST:

- Resources need to be addressable
- Uses a uniform, constrained interface
- Focused on representing the problem
- Communicate Statelessly
- Hypermedia As The Engine Of Application State (HATEOAS)

Addressable resources means that any resource should be referenced by a unique address that is completely self-contained. Doing this allows us to send small amounts of information to express complex objects (i.e images) and - if the other user has the data they don't need to download the entire thing.

REST often uses URI's to represent the various resources.

Uniform Constrained Interface means that the interface is used in the same way everywhere and is only used for its intended purposes. (I.E using the right HTTP methods.) This also allows for caching of repeated uses of an interface method.

Stateless means that there is no session maintained over multiple visits, instead we store this data as a cookie. We do this so that we can easily control what operations are valid to allow data changes to be atomic and isolated.

We also focus on using a uniform constrained interface to ensure that transactions are consistent and durable.

In REST we focus on being representation oriented with communication. This means we communicate between the server and the client using MIME types. These are generic representations of data like text/plain, text/html, application/xml and application/json. This allows consumers to negotiate for datatypes without adding a lot of extra steps to generate thousands of different layouts of data. (universal few datatypes).

We also use stateless communication so that the server can change without influencing the affects on the client (as the other server might not have the same state as the original server after changing).

Finally we consider HATEOAS [hypermedia as the engine of application state]. This means we should allow user traversal with hypermedia links as opposed to getting users to find the links themselves.

This allows the server to track where the user may go next and cache idealistic data?

HTTP Methods:

GET is used when the client requests something from the server.

HEAD is used when the client wants the metadata.

POST is used for processing form data.

PUT is used to update some data on the serverside.

DELETE is used to remove a resource on serverside.

OPTIONS us used to request the available options from serverside.

TRACE is used to track what requests happened.

1xx responses are informational

2xx response codes are successful

3xx response codes are redirection

4xx response codes are client errors.

5xx response codes are server errors.

HTTP is very useful because of it's stateless architecture (and controlled state cookies)

It is also very useful because it has negotiable content, that is that a client can request different kinds of content types and the server will return the best one that it can for the browser.

Lecture 4

Topics:

- JAX-RS
 - Programming Model with JAX
 - REST with JAX
- Example (how to code in JAX)
- JAX API