

MAD Adv II

Closures

- Closure is a way of implementing private class state in javascript by playing on variable scope

```
function add() {  
  var counter=0;  
  counter += 1;  
  console.log(counter);  
}
```

```
add();  
add();
```

```
var add = (function () {  
  var counter = 0;  
  return function () {  
    counter += 1;  
    console.log(counter);  
    return counter;  
  };  
})();
```

```
add();  
add();
```

Closures As Function Factories

```
function notifyUser(email) {  
    return function(text) {  
        console.log("Sending '"+text+"' to "+email);  
    };  
}
```

```
var notifyWill = notifyUser('wberger@lo.com');  
var notifyKelly = notifyUser('kperry@lo.com');
```

```
notifyWill("Hello, you have a new join request");  
notifyKelly("Hello you are late to work!");
```

Closures As Modules

```
var counter = (function() {  
  var privateCounter = 0;  
  function changeBy(val) {  
    privateCounter += val;  
  }  
  return {  
    increment : function() {  
      changeBy(1);  
    },  
    decrement : function() {  
      changeBy(-1);  
    },  
    value : function() {  
      return privateCounter;  
    }  
  };  
})();
```

```
console.log(counter.value());  
counter.increment();  
counter.increment();  
console.log(counter.value());  
counter.decrement();  
console.log(counter.value());
```

Singletons With Closures

```
var mySingleton = (function() {  
    var instance;  
    function init() {  
        function privateMethod() {  
            console.log("I am private");  
        }  
        var privateVariable = "Im also private";  
        var privateRandomNumber = Math.random();  
        return {  
            publicMethod : function() {  
                console.log("The public can see me!");  
            },  
            publicProperty : "I am also public",  
            getRandomNumber : function() {  
                return privateRandomNumber;  
            }  
        };  
    };  
};  
  
return {  
    getInstance : function() {  
        if (!instance) {  
            instance = init();  
        }  
        return instance;  
    }  
};  
})();
```

Closures & Objects

```
var cars = [{brand: 'Fiat', model: 'Punto'}, {brand: 'Hyundai',  
model: 'Santro'}, {brand: 'Maruti', model: 'Swift'}];  
for (var i = 0; i < cars.length; i++) {  
    cars[i].printLocation = function() {  
        console.log(this.model + " is in location: " + i);  
    };  
}  
  
cars[0].printLocation();
```

Fix the looping problem

```
for (var i = 0; i < cars.length; i++) {  
    cars[i].printLocation = function(x) {  
        return function() {  
            console.log(this.model + " is in location: " + x);  
        };  
    }(i);  
}  
  
cars[0].printLocation();
```

Lets Try It

- Create a collection class in Javascript with closure to hold the private state. Provide only the add and size methods

Closures

- Closures are a basic part of many languages today: Javascript, Python, Ruby (even java in a way)
- Closures pave the way to creating new language constructs without changing language specs. Consider this:

```
interface OneArg<A>{
    void invoke(A arg);
}

<T> void forEach(Collection<T> c, OneArg<T> block){
    Iterator<T> it = c.iterator();
    while(it.hasNext()){
        block.invoke(it.next());
    }
}

forEach(c, data->{
    System.out.print(data);
    System.out.println("Total size: "+c.size());
});

forEach(c, new OneArg() {
    public void invoke(Object data) {
        System.out.print(data);
        System.out.println("Total size: "+c.size());
    }
});
```

Build A Contextual Try

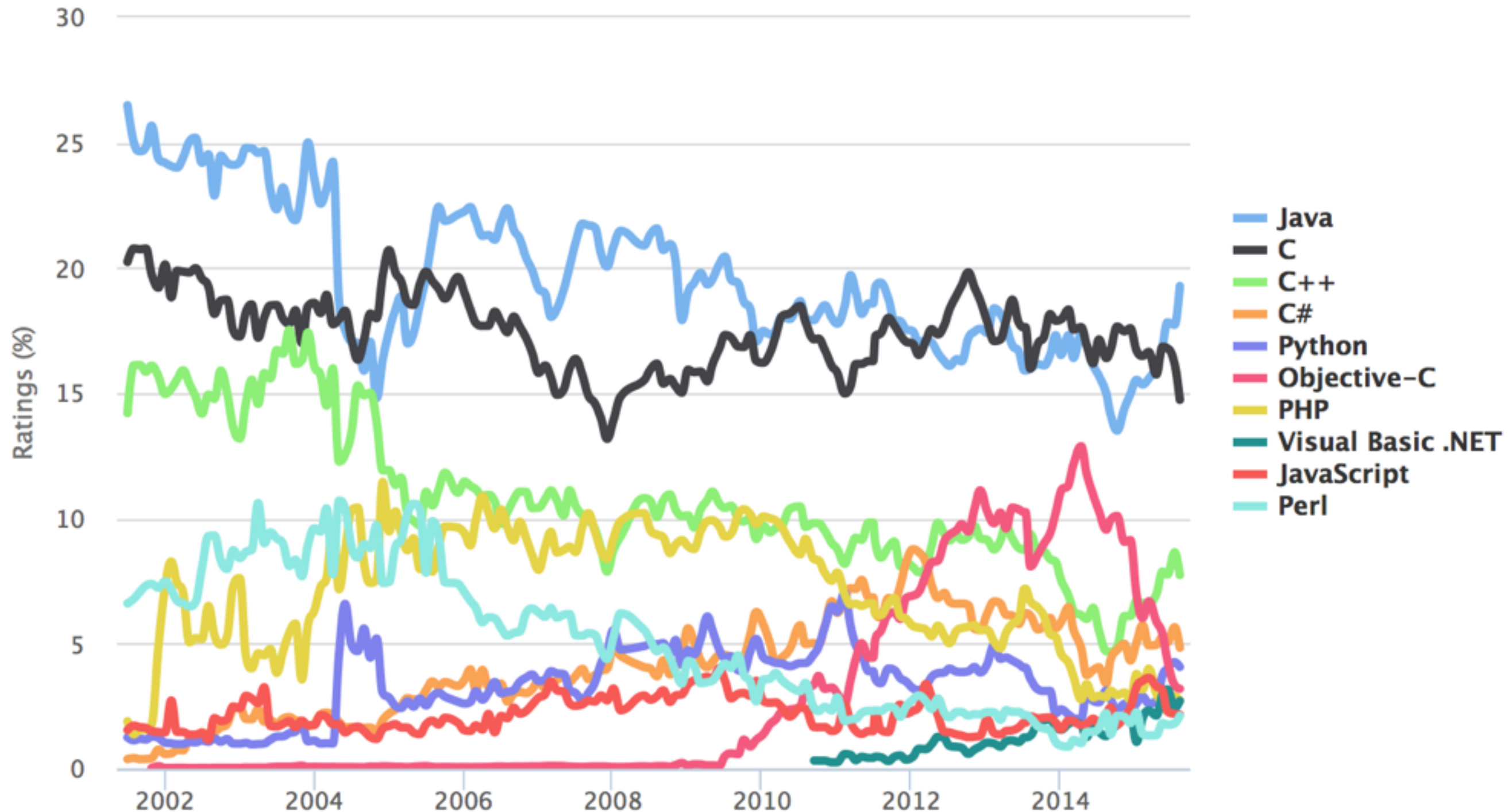
```
with(con, new OneArg<Connection>() {  
    public void invoke(Connection con) {  
        //Do something with connection  
    }  
});  
with(con, connection->{  
    connection.doSomething()  
});
```

```
void with(Connection con, OneArg<Connection> block) {  
    try {  
        block.invoke(con);  
    } finally {  
        if(con!=null)  
            try {  
                con.close();  
            } catch (SQLException e) {}  
    }  
}
```

Over the Years

TIOBE Programming Community Index

Source: www.tiobe.com



Lambda Functions

- Typical patterns with anonymous classes and functional interfaces

```
JButton testButton = new JButton("Test Button");
testButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent ae) {
        System.out.println("Click Detected by Anon Class");
    }
});
```

```
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

Lambdas

- Try these in the IDE:

```
m = (int x, int y) -> x + y;
```

```
n = () -> 42;
```

```
o = (String s) -> { System.out.println(s); };
```

- Launch a thread using a runnable lambda
- Sort a list using the comparable lambda

Type Inferencing and Parentheses

- We don't have to specify the type of arguments in a lambda expression.

```
(Float m)->{  
    //method impl  
}
```

```
(m)->{  
    //method impl  
}
```

```
m->{  
    //method impl  
}
```

Expression For Body

- If the interface method we are making a lambda has a ret val, the lambda need not return it directly
 - An expression in lambda is treated as a return value

```
interface SomeInterface{
    public Integer methodOne(Float m);
}

class SomeClass {

    public void someMethod(SomeInterface i){
        //Do something
    }

    public static void main(String[] args) {
        SomeClass x = new SomeClass();
        x.someMethod((m)-> (int)(m*m));
    }
}
```

Single statements are always not expressions

```
x.someMethod((m)-> if(m>5) return 3; else return 5;);
```

Thinking

```
Arrays.sort(strings, (a,b)->a.length-b.length);
```

- Whats Really Happening
 - A shortcut way of representing an instance of a class that implements `Comparator<T>`. Body of compare method is provided after `->`
- How we usually think of it
 - We passed a comparison function or expression

Functional Interfaces

- `java.util.function` package has a lot of functional interfaces
 - Predicate: A property of the object passed as argument
 - Consumer: An action to be performed with the object passed as argument
 - Function: Transform a T to a U
 - Supplier: Provide an instance of a T (such as a factory)
 - UnaryOperator: A unary operator from $T \rightarrow T$
 - BinaryOperator: A binary operator from $(T, T) \rightarrow T$

Scope

- Lambda functions don't introduce another scoped class.
- `this` in lambda function refers to the class in which it is used
- Lambda function has access to local variables and methods of the calling class

Method References

- You use lambda expressions to create anonymous methods.
- Sometimes, however, a lambda expression does nothing but call an existing method.
- In those cases, refer to the existing method by name.

```
Arrays.sort(rosterAsArray, myComparisonProvider::compareByName);  
Arrays.sort(stringArray, String::compareToIgnoreCase);  
Set<Person> rosterSet = transferElements(roster, HashSet::new);
```

Streams

- Streams are a way to sequentially access a list with lot of useful functions
- Streams are faster and memory efficient ways of processing lists
 - Lazy Evaluation
 - Automatic Parallelization
 - infinite unbounded streams
- Streams do not store data. Just programmatic wrappers on data structures

Streams

- Support many high performance operations expressed with lambdas expressed sequentially or in parallel
- Streams do not modify the underlying data structure

```
Stream s = Stream.of(empids)  
                .map(dao::findEmpById)  
                .filter(e->e.getSalary()>10000);
```

```
Employee richEmployee = s.findFirst()  
                        .orElse(null)
```

Streams

- Designed for lambdas
- Do not support indexed access
- However can generate arrays
- Lazy evaluation - only needed amount of data is analyzed
- Parallel execution without creating threads
- Tied with a generator function, streams can be infinite

Important Thing about Streams

- When you make or transform a Stream, it does not copy the underlying data. Instead it just builds a pipeline of operations. How many times this pipeline is invoked will depend on what we do with the stream in the end. Such as find first element or sort etc.
- Ways to create stream
 - `list.stream()`
 - `Stream.of(array)`
 - `Stream.of(val1, val2,..)`

Try this

- Send an email to 100 customers present in a list
- Do not send emails to customers with Yahoo mail id
- Now eliminate any embedded links in the email content
- Count successfully sent emails

Stream Issues

- Try this

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
    .distinct()
    .limit(10)
    .forEach(System.out::println);
```

- And this

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
    .parallel()
    .distinct()
    .limit(10)
    .forEach(System.out::println);
```

- And this

```
IntStream.range(1, 5)
    .peek(System.out::println)
    .peek(i -> {
        if (i == 5)
            throw new RuntimeException("bang");
    });
```

Samples

```
public ProductivityReport computeProductivity(long projectId,
LocalDateTime startDate) {
    List<SvnCheckin> projectCheckins = _svnCheckinRepository
        .findByProductIdAndCommitTimeGreaterThanAndJobResultIn(projectId, startDate,
            ServiceStatus.COMPLETED);

    Map<String, List<SvnCheckin>> checkinsGroupedByDev =
        projectCheckins.stream()
            .collect(Collectors.groupingBy(SvnCheckin::getAuthor));

    List<DeveloperProductivityReport> devReports =
        checkinsGroupedByDev.entrySet().stream()
            .map(this::createDeveloperProductivityReport).collect(Collectors.toList());

    return new ProductivityReport(devReports);
}
```

Samples

```
private DeveloperProductivityReport createDeveloperProductivityReport(
    Map.Entry<String, List<SvnCheckin>> developerCheckins) {
    long linesAccepted = developerCheckins.getValue().stream()
        .filter(checkin -> checkin.getJobResult() == JobResult.ACCEPTED)
        .mapToLong(e -> e.getLinesAdded() + e.getLinesRemoved() +
e.getLinesModified()).sum();

    return new
DeveloperProductivityReport.DeveloperReportBuilder().withAuthor(developerChecki
ns.getKey())
        .withCheckins(developerCheckins.getValue()).withLinesAcceptedByFire
wall(linesAccepted)
        .createAuthorReport();
}
```

Optional

- Methods like `findFirst` return an `Optional`
- `Optional<SomeClass> val = Optional.of(x)`
- `Optional<SomeClass> val = Optional.empty()`
- `value.get(), value.orElse(otherVal),
value.orElseGet(supplier), value.isPresent()`

- Examples

```
job.setJobType(Optional.ofNullable(firewallJob.getJobType()).map(
JobType::getJobTypeName).orElse("")))
```

```
private boolean isFwKilled(Optional<BuildJob> buildJob,
Optional<MetricsJob> metricsJob, Optional<UnitTestJob>
unitTestJob) {
    _logger.debug("Metrics job details in isFwKilled: {}",
metricsJob.map(job -> job.getStatusDetails()).orElse(""));
    return isJobKilled(getBuildStatus(buildJob)) ||
        metricsJob.orElse(new
MetricsJob()).isKilledAndNotAborted() ||
        isJobKilled(getUnitTestStatus(unitTestJob));
}
```

Generics

- ◆ Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods
- ◆ Stronger type checks at compile time
 - ◆ Fixing compile-time errors is easier than fixing runtime errors

Elimination Of Casts

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

Generic Class

```
public class Box {  
    private Object object;  
    public void set(Object object) { this.object = object; }  
    public Object get() { return object; }  
}
```

```
public class Box<T> {  
    private T t;  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

```
Box<Integer> integerBox = new Box<Integer>();
```

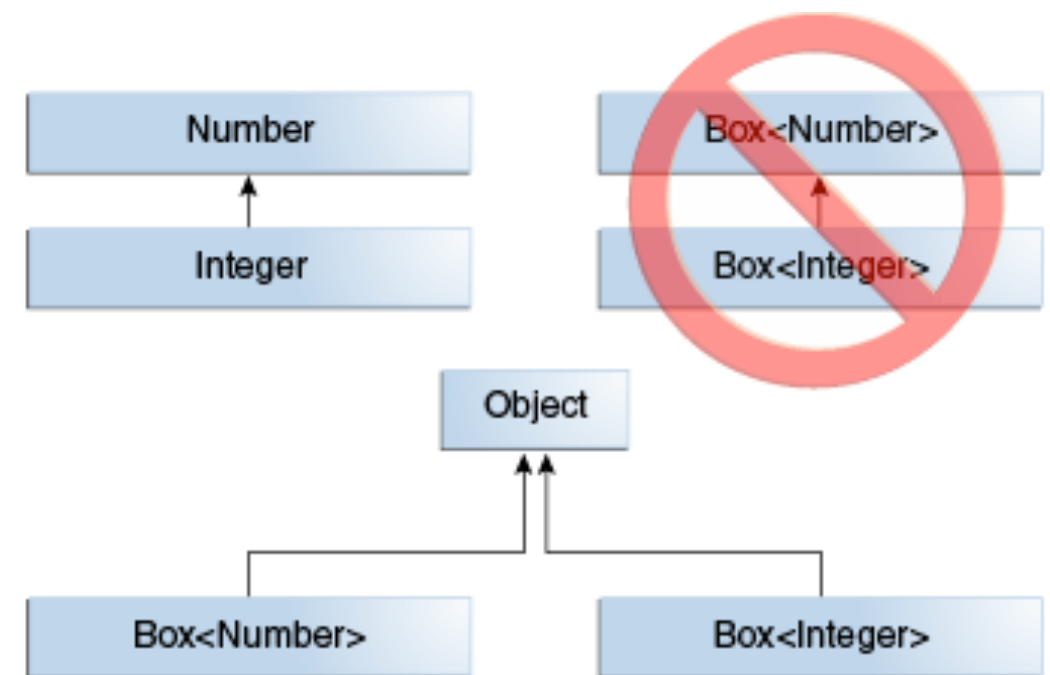

Generic Methods

```
public static <K, V> boolean join(K p1, V p2) {  
    return p1.toString() + " " + p2.toString();  
}
```

```
Util.<Integer, String>join(p1, p2);
```

Inheritance With Generics

- ◆ Generics are implemented at the compiler level using erasure
 - ◆ Runtime system does not know about generics at all
 - ◆ Hence...
 - ◆ Instanceof wont work with generics



Effect Of Erasure

```
List<Integer> mn = new ArrayList<>();  
mn.add(5);  
Collection n = mn; // A raw type - compiler throws an unchecked warning  
n.add("Hello");  
Integer x = mn.get(1); //Problem! Causes a ClassCastException to be thrown.
```

Restrictions

- ◆ Cant use primitive type params
 - ◆ `List<int> k;`
- ◆ Cannot Create Instances of Type Parameters
 - ◆ `public static <E> void append(List<E> list) {`
 - ◆ `E elem = new E(); // compile-time error`
- ◆ Cannot Declare Static Fields Whose Types are Type Parameters
 - ◆ `public class MobileDevice<T> {`
 - ◆ `private static T os;`
- ◆ Cannot Use Casts or instanceof with Parameterized Types
 - ◆ `if (list instanceof ArrayList<Integer>) { // compile-time error`
- ◆ Cannot Create Arrays of Parameterized Types
 - ◆ `List<Integer>[] arrayOfLists = new List<Integer>[2];`
- ◆ Cannot Create, Catch, or Throw Objects of Parameterized Types
 - ◆ `catch (T e) {`
- ◆ Cannot overload
 - ◆ `public void print(Set<String> strSet) { }`
 - ◆ `public void print(Set<Integer> intSet) { }`

Promises

- Promises are objects that have a “then” method
- Unlike node functions, which take a single callback, the then method of a promise can take two callbacks: a success callback and an error callback.
- When one of these two callbacks returns a value or throws an exception, then must behave in a way that enables stream-like chaining and simplified error handling.

Without Promises

```
function doSomething(callback) {  
  setTimeout(function() {  
    console.log("2 seconds up");  
    callback(null, "2 seconds up");  
  }, 2000);  
}
```

```
function doSomethingElse(callback) {  
  setTimeout(function() {  
    console.log("2 more seconds up");  
    callback(null, "2 more seconds up");  
  }, 2000);  
}
```

```
doSomething(function() {  
  doSomethingElse(function() {  
    console.log("Should be 4 seconds now!");  
  });  
});
```

With Promises

```
var Promise = require("bluebird");
function doSomethingWithPromise() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      console.log("2 seconds up");
      resolve("2 seconds up");
    }, 2000);
  });
}

function doSomethingElseWithPromise() {
  return new Promise(function(resolve, reject) {
    setTimeout(function() {
      console.log("2 more seconds up");
      resolve(null, "2 more seconds up");
    }, 2000);
  });
}

doSomethingWithPromise()
  .then(function() {
    return doSomethingElseWithPromise();
  })
  .then(function() {
    console.log("Should be 4 seconds now!");
  });
```

Promisification

```
var doSomethingWithPromise = Promise.promisify(doSomething);  
var doSomethingElseWithPromise = Promise.promisify(doSomethingElse);
```

```
doSomethingWithPromise()  
  .then(function() {  
    return doSomethingElseWithPromise();  
  }).then(function() {  
    console.log("Should be 4 seconds now!");  
  });
```


Changes With Promises

//Regular Node code

```
fs.readFile(file, function(err, res) {  
    if (err) handleError();  
    doStuffWith(res);  
});
```

//With promises

```
fs.readFile(file).then(function(res) {  
    doStuffWith(res);  
}, function(err) {  
    handleError();  
});
```

Promises...

- Promises are kept in future too

```
var filePromise = fs.readFile(file);  
//do more stuff... and then  
filePromise.then(function(res) { //process data in res  
});
```

- Promises are good for multiple things

```
filePromise.then(function(res) { uploadData(url, res); });  
filePromise.then(function(res) { saveLocal(url, res); });
```

- Returns from promises become promises

```
var firstLinePromise = filePromise.then(function(data) {  
    return data.toString().split('\n')[0];  
});
```

```
var beginsWithHelloPromise = firstLinePromise.then(function(line) {  
    return /^hello/.test(line);  
});
```

Error Handling With Promises

```
function readProcessAndSave(inPath, outPath) {
  var filePromise = fs.readFile(inPath);
  var transformedPromise = filePromise.then(function(content) {
    return service.transform(content);
  });
  var writeFilePromise =
    transformedPromise.then(function(transformed) {
      return fs.writeFile(otherPath, transformed)
    });
  return writeFilePromise;
}
readProcessAndSave(file, url, otherPath).then(function() {
  console.log("Success!");
}, function(err) {
  console.log("Error: ", err);
});
```

Promisify

- Bluebird.js is a promise library that allows for converting regular node libraries to support promises
- For this to work, node libraries should follow the convention of `callback(err, successVal)`

```
Promise.promisifyAll(CommandProcessorAsyncE.prototype);
```

```
//Method will be called in context of cpasyncce
```

```
var cp = Promise.promisify(cpasyncce.processCommand, cpasyncce);
```

```
var fs = require("fs");
```

```
Promise.promisifyAll(fs);
```

Chaining And Error Handling

```
return fs.readFileAsync(inPath)
    .then(service.transform)
    .then(function(transformed) {
        return fs.writeFileAsync(otherPath, transformed)
    }).caught(function(e){
        console.log("Error in operations");
        console.log(e.stack);
    }).lastly(function(){
        console.log("Executed finally");
    });
```

Joins

- We can co-ordinate the result of multiple promises using join.

here is an example:

```
function doSomething(callback) {
  setTimeout(function() {
    console.log("2 seconds up");
    callback(null, "2 seconds up");
  }, 2000);
}
function doSomethingElse(callback) {
  setTimeout(function() {
    console.log("5 seconds up");
    callback(null, "5 seconds up");
  }, 5000);
}
```

```
var doSomethingWithPromise = Promise.promisify(doSomething);
var doSomethingElseWithPromise = Promise.promisify(doSomethingElse);
```

```
Promise.join(doSomethingWithPromise(), doSomethingElseWithPromise(), function(val1, val2){
  console.log("Joined: "+val1);
  console.log("Joined: "+val2);
});
```

Print all files in directory /Users/folder1 and /Users/folder2 together using our command processor

Map

- Map allows an array to be transformed to another array using a function that can transform an individual element using an async promise, using a specific concurrency

```
var fileNames = ["package.json", "package2.json"];
Promise.map(fileNames, function(fileName) {
    return fs.readFileAsync(fileName)
        .then(JSON.parse)
        .caught(SyntaxError, function(e) {
            e.fileName = fileName;
            throw e;
        });
}, {concurrency: 1}).then(function(parsedJSONs) {
    console.log(parsedJSONs);
}).caught(SyntaxError, function(e) {
    console.log("Invalid JSON in file " + e.fileName + ": " + e.message);
});
```

Compress files in a directory using this map mechanism

Reduce

- Calls a function with each promise return value to arrive at a cumulative aggregate value.
- Reduction function is called as soon as any promise result is ready (unlike join/all)

```
Promise.reduce(  
  [ promise1, promise2 ],  
  function(reducedResult, result) {  
    return reducedResult + result.value  
  }, 0).then(function(reducedResult) {  
    console.log("Final reduced result: " + reducedResult);  
  });
```

Write a new folder size command in CommandProcessor using the reduction function. Use fs.readAsync method

Filter

```
Promise.filter([fs.readFileAsync("/Users/maruthir/ir.sql"),fs.readFileAsync("/Users/maruthir/maruthi.pem")],function(content){
    console.log("Filter func");
    if(content.length>5000)
        return false;
    else
        return true;
}).then(function(args){
    console.log(args);
});
```

Modify the list command to accept a file pattern and list files only with that pattern

Functional Reactive Programming

- Functional reactive programming (FRP) is a programming paradigm for reactive programming (asynchronous dataflow programming) using the building blocks of functional programming (e.g. map, reduce, filter)

Building Blocks

- Map
 - Transform operations that change an object
- Filter
 - Decision operation that returns boolean to determine if the object needs to be in the output collection
- Reduce
 - Aggregation operation that works on a stream of data and produces a single property

Where is this useful

- When there are too many events from diverse sources manipulating too many state fields
- We bundle events into different pipes. And attach functional programs to these pipes to perform actions.
- Events evolve with map, filter and reduce

Event Stream Concept

- First concept of FRP is assemble happenings in the system as a stream of events
 - Events can come from a promise (A single event and end of stream): `Bacon.fromPromise(promise)`
 - From Node.js event emitters:
`Bacon.fromEventTarget(eventEmitter, eventName)`
 - Single event from a function that takes a callback:
`Bacon.fromNodeCallback(f)`
 - `Bacon.fromPoll(interval, f)`: `f` should return `Bacon.next` or `Bacon.end`. `f` is called in intervals

On the Browser

- Events can also come from jquery bound controls

```
$("#input[name='loginname']").asEventStream("keyup")
```

- Or from an array of data

```
return Bacon.fromArray(data.data);
```

- Once we create a stream, we can listen to the stream using onValue function

```
myStream.onValue(function(streamData) {  
    console.log(streamData);  
});
```

Lets Try It

- Create an input box, listen to keyup events and log them to console

Transforming Events

- Map functions are used to transform events on streams
- `stream.map(function(data){ return transformedData})`

Lets Try It

- Transform the event objects coming out of the keyup stream to the text value contained in the event and see how the listening on the stream goes

```
var str = $(event.target).val()
```

- In the value listener, update a div next to the user input box to indicate if the username is valid or not (yes/not)
- Transform the text value to Yes/No values depending on if the text is greater than 8 chars or not

Properties

- Properties are very similar to streams except that they have a current value & initial value.
- Properties are result of “reduce” operations on some stream
- `property.sample(interval)` - get current value at certain intervals
- `property.sampledBy(stream)` - get current value of property every time there is an event on the stream

Lets Try It

- Convert the user stream to a property by calling - `toProperty("No")`
- This should provide an initial value to the status display

Combining Streams With Properties

- A value from a stream can be combined with a value from a property to arrive at a new stream
- `stream.combine(property, function(a,b){ return a+b;})`
- Create another text field for email address input and create a yes/no property for it after validating it has the @ character in it.
- Combine this property with the username yes/no property to arrive at a joint validity decision property

Streams from Ajax

- JQuery ajax calls return deferred objects

```
$.ajax({ url : "isUserValid.txt"}).done(function(result)  
{ console.log(result)});
```

- This deferred object can be part of a stream
- Lets check if the user is valid on every keyup event. To the earlier created username property, create a map function that returns ajax responses.
- Base the “validity of field” decision on the Ajax response being true or false

FlatMaps

- flatMap is a function that creates a stream for every event in the incoming stream
- `stream1.flatMap(function(event){ return subStream});`
- Output is a single stream that has all elements of the sub-streams flattened

Lets Try It

- Load the players from players.json provided in the server.
- Convert an ajax request into a stream
`Bacon.fromPromise($.ajax({ url : "players.json"}));`
- Call flatMap on resultant stream to make the data a stream `Bacon.fromArray(data.data);`
- On the stream value handler, append the user to a div

Filter Operation

- Streams can be filtered to eliminate the events not needed
- `stream.filter(function(eventData){ return boolean})`
- Modify the previous user listing to eliminate retired players from the display.

Reduce Operation

- Scan function is used to perform reduce operation based on previous stream values:
`scan(initialValue,function(prev,current){return
accumulate(prev and current);});`
- Display a count of players by doing a scan operation on the filtered stream

Manipulate Streams - Map, Filter, Reduce

- Methods on streams: `.onValue`, `.onError`, `.onEnd`,
- `.map(function(value){})` - Converts events in this stream using the provided function
- `.map(property)` - Puts property into stream for every event on stream
- `.filter(property)`

Micro Services

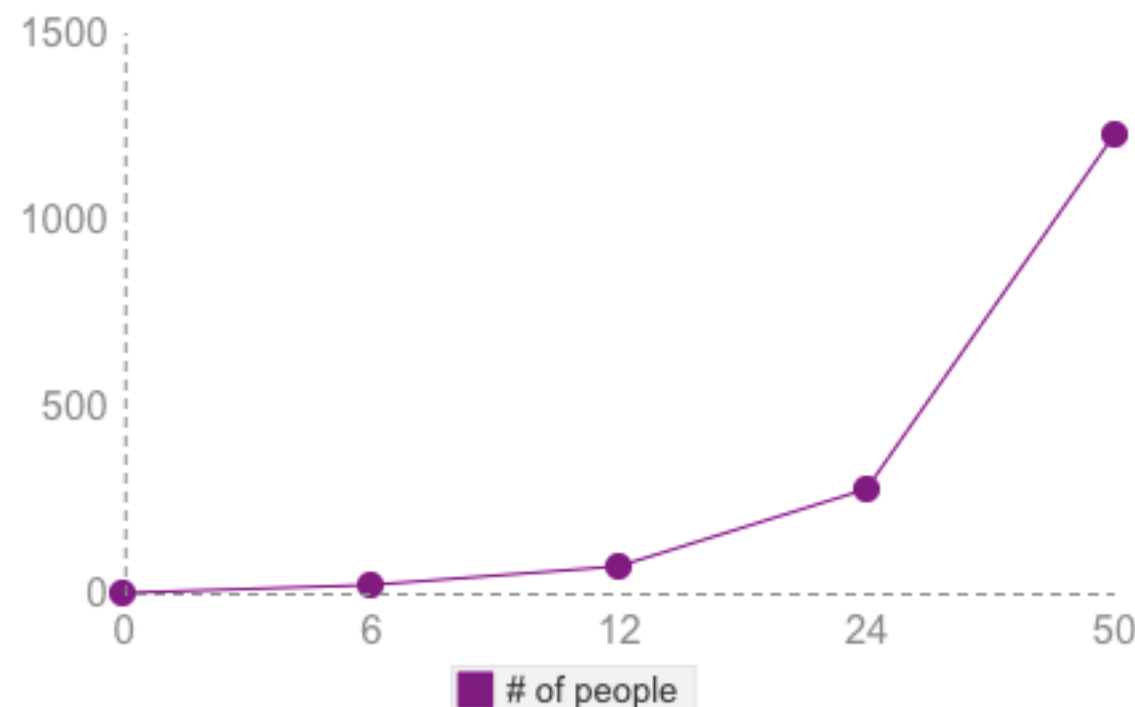
- Small (100? 1000? 5000? LOC)
- UI/ Without UI?
- Lightweight - run several per box
- Independent development/deployment
- Stateless - everything in DB. Can use chaos monkey
- Monitored

Conveys Law

- organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations
- Conway's law is a valid sociological observation.
- For separate software modules to interface correctly, the designers and implementers of each module must communicate with each other as loosely as tightly as desired by the modules themselves.

Two Pizza Teams

- Amazon founder and CEO Jeff Bezos created the concept that teams shouldn't be larger than what two pizzas can feed
- Its like a dinner gathering. If u have to order more than two large pizzas, u r forming clusters of conversations
- $n(n-1)/2$ is the number of connections in team



Team Scaling Fallacy

- This is the tendency for people “to increasingly underestimate task completion time as team size grows,”
- One warton university research (http://www.opim.wharton.upenn.edu/~kmilkman/2012_OBHDPb.pdf) shows that when tasked to build the same Lego figure, two-person teams took 36 minutes while four-person teams took 52 minutes to finish — over 44% longer.

Back to Micro Services

- 1 micro service = 1 team
- 1 team = may be >1 micro service
- So a micro service cant be anything thats larger than what can be built with 5 to 6 people and deployable in a week or two week's time

YOU build, YOU run

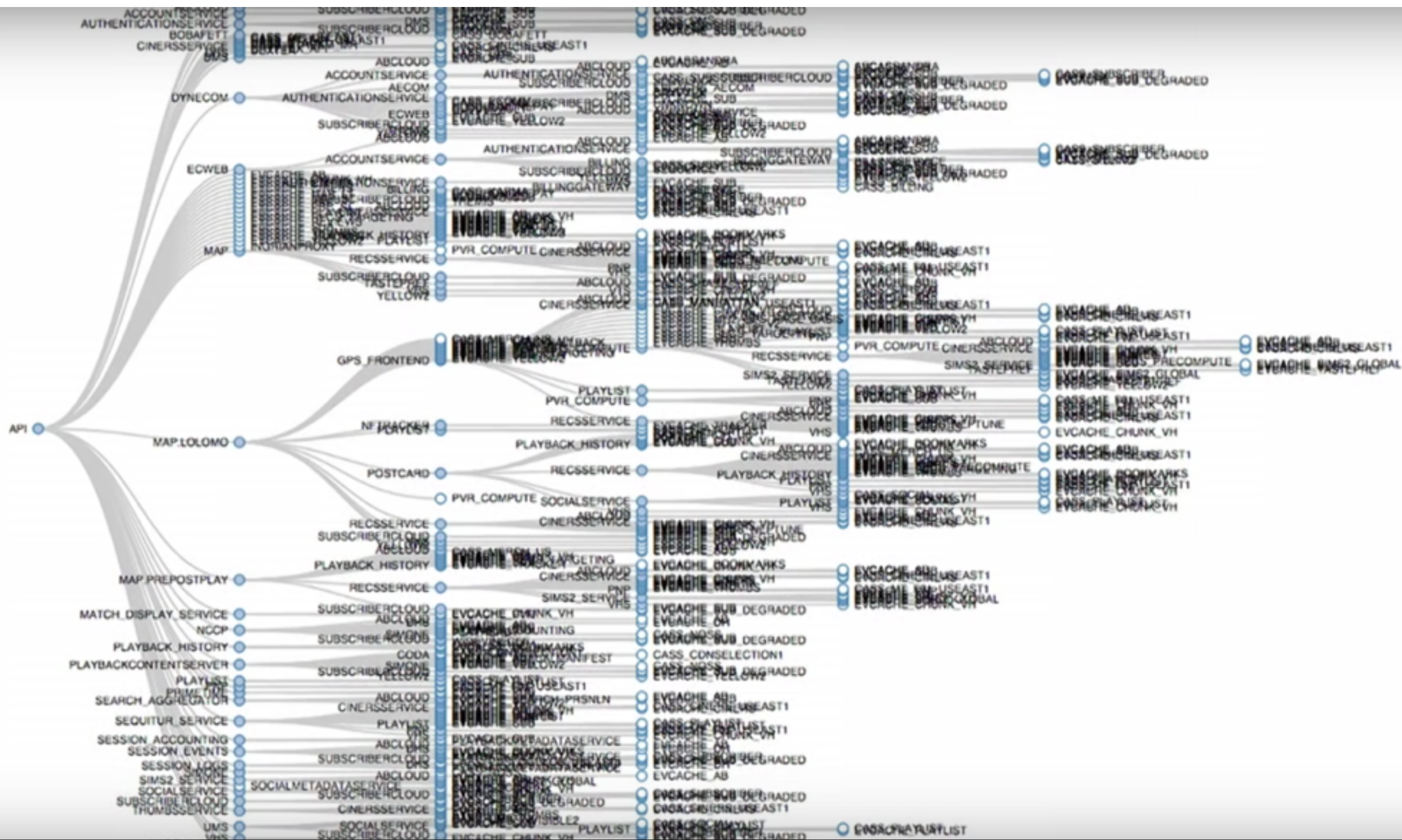
- Micro services is an organizational change.
- The team that builds it also runs it on production with little infrastructure help.
- No passing the buck, no separate admin/dev support mess and since you feel the heat, you will also be more responsible!

Big Debate

- Should it have an UI or not?
- If it does not have UI, how is it different from SOA
- How do we handle sandboxing on a browser page?
 - Javascript sandboxing, CSS enclosures.
 - Angular JS concept of apps, controllers and services demo.

Netflix Case Study

- 2008 - large data center with a big .war file!
- 2012 - Amazon cloud with micro services
- Amazon prime is a competitor, still they decided to use Amazon cloud. Adrian Cockcroft says, lets at the least have the same advantage that Amazon will get.



Netflix ecosystem

100s of microservices

1000s of daily production changes

10,000s of instances

100,000s of customer interactions per minute

1,000,000s of customers

1,000,000,000s of metrics

10,000,000,000 hours of streamed

10s of operations engineers

Adrian's recipe for Micro Services

- you can update the services independently; updating one service doesn't require changing any other services
- If you have a bunch of small, specialized services but still have to update them together, they're not microservices because they're not **loosely coupled**.
- Cant even talk to same db. You need to split the database up and denormalize it.
- You need to add a tool that performs master data management (MDM) by operating in the background to find and fix inconsistencies.

Bounded Context

- bounded contexts comes from the book Domain Driven Design by Eric Evans. A microservice with correctly bounded context is self-contained for the purposes of software development.
- microservices and its peers interact strictly through APIs and so don't share data structures, database schemata, or other internal representations of objects.

Immutable Infrastructure Principle

- If you need to add or rewrite some of the code in a deployed microservice that's working well, the best approach is usually to create a new microservice version for the new or changed code
- In Cockcroft's experience it is much more common to realize you should split up a microservice because it's gotten too big.

Building Micro Services

- Do a separate build for each microservice, so that it can pull in component files from the repository at the revision levels appropriate to it.
- Deploy in a standard type of container such as docker so the service deployment process is standard across the organization
- Servers are stateless “cattle” instead of being “pets”

Pros

- Isolation brings better availability
- Independent delivery cycle for each service
- Decentralised Governance

Cons

- Complexity of distributed systems
- Operational overhead in devops
- Service versioning and testing issues
- Needs load balanced service discovery for critical services

Failures

- If the probability of an app being up is 99.99%
 - Breaking it down into 30 services with an uptime of each 99.99% gives us a system that's up only 99.97%
- This is one of the reasons microservices have to be fully contained independently degradable end to end functionalities.
 - Not parts of the system that all need to work for the full system to be up.
 - This way individual failures break part of the system only.

Failures

- For critical services we need load balanced discovery
- Netflix created and open sourced ribbon
 - Ribbon is a client side IPC library that is battle-tested in cloud. It provides the following features
 - Load balancing
 - Fault tolerance
 - Multiple protocol (HTTP, TCP, UDP) support in an asynchronous and reactive model
 - Caching and batching

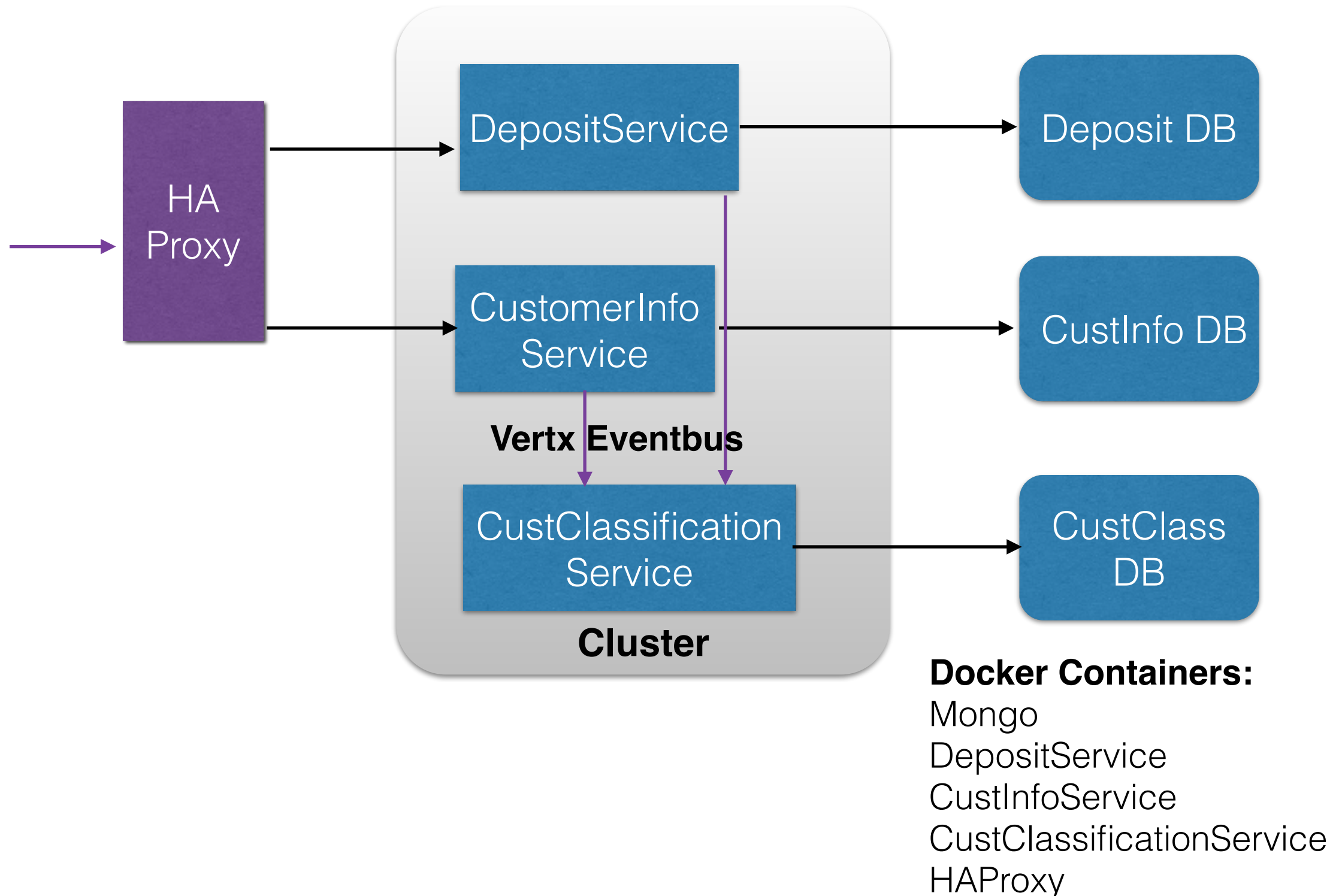
Create the following

- Rest API to perform deposits into account.

`"/account/:accountNumber"` - add and get account with balance info
`/account/:accountNumber/money` - add money

- Rest API to get and create account holder info
- Classify customers as A, B or C based on their deposits, age and profession

Architecture



- Vertx clustering is enabled with hazelcast dependency and cluster.xml. Start vertx with -cluster option in dockerfile

```
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-mongo-client</artifactId>
  <version>3.2.1</version>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-core</artifactId>
  <version>2.7.4</version>
</dependency>
<dependency>
  <groupId>io.vertx</groupId>
  <artifactId>vertx-hazelcast</artifactId>
  <version>3.2.1</version>
</dependency>
```

HA Proxy

- Needs a config file

```
frontend http-in
```

```
    bind *:80
```

```
    default_backend deposits
```

```
    acl customer path_sub /customer
```

```
    use_backend custinfo if customer
```

```
backend deposits
```

```
    server server1 192.168.99.100:8080 maxconn 32
```

```
backend custinfo
```

```
    server server2 192.168.99.100:8090 maxconn 32
```

- and dockerfile

```
FROM haproxy:1.5
```

```
COPY haproxy.cfg /usr/local/etc/haproxy/haproxy.cfg
```


Container scripts

```
mvn package
docker build -t deposit .
docker run -i -p 8080:8080 -p 5701:5701 --name r_deposit deposit
```

```
mvn package
docker build -t custclass .
docker run -i -p 5703:5703 --name r_custclass custclass
```

Stop all script:

```
docker stop r_custclass
docker stop r_deposit
docker stop r_custinfo
docker rm r_custclass
docker rm r_deposit
docker rm r_custinfo
docker rmi custclass
docker rmi custinfo
docker rmi deposit
```

Distributed Systems

- What is distributed?
- Why it is distributed?
- What are the problems?
- How do we solve them?
- What are the tradeoffs?

What is distributed?

- Computation - least troublesome
- Storage
 - Replication
 - Sharding
 - A combination

Why is it distributed

- Computation distribution is to support compute loads
- Also to add fault tolerance
- Stateless compute logic is easy to distribute and scale

Why is it distributed

- Storage distribution is to do one of the following:
 - Improve load capacity
 - Improve redundancy - backups are no longer possible on web scale
 - Improve concurrency

What are the problems

- Compute Systems
 - Ordering of events
 - Session failovers in stateful compute distribution
 - Distributed service discovery
 - Cluster wide rippling deployment

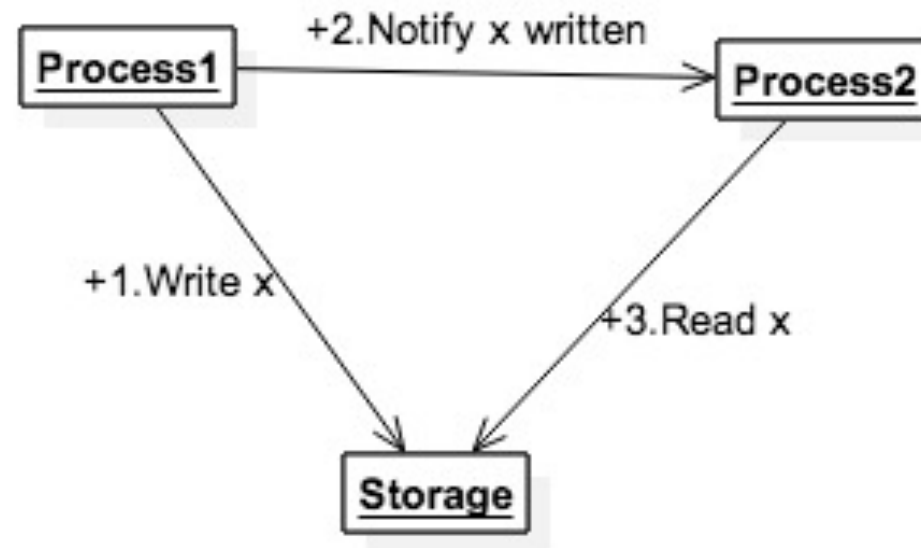
What are the problems

- Storage Distribution
 - Consensus
 - CAP problems
 - Shrad location
 - Cross shrad queries
 - Write performance
 - Read performance

What are the solutions

- Consensus: Protocols like paxos and raft
- <http://thesecretlivesofdata.com/raft/>

Event Ordering Problem



- Event 1 and 3 can arrive at storage node out of order or at the same time

Event ordering

- Lamport clocks
- The algorithm of Lamport timestamps is a simple algorithm used to determine the order of events in a distributed system.
- As different nodes or processes will typically not be perfectly synchronized, this algorithm is used to provide a partial ordering of events with minimal overhead - Leslie Lamport.

Lamport Clocks

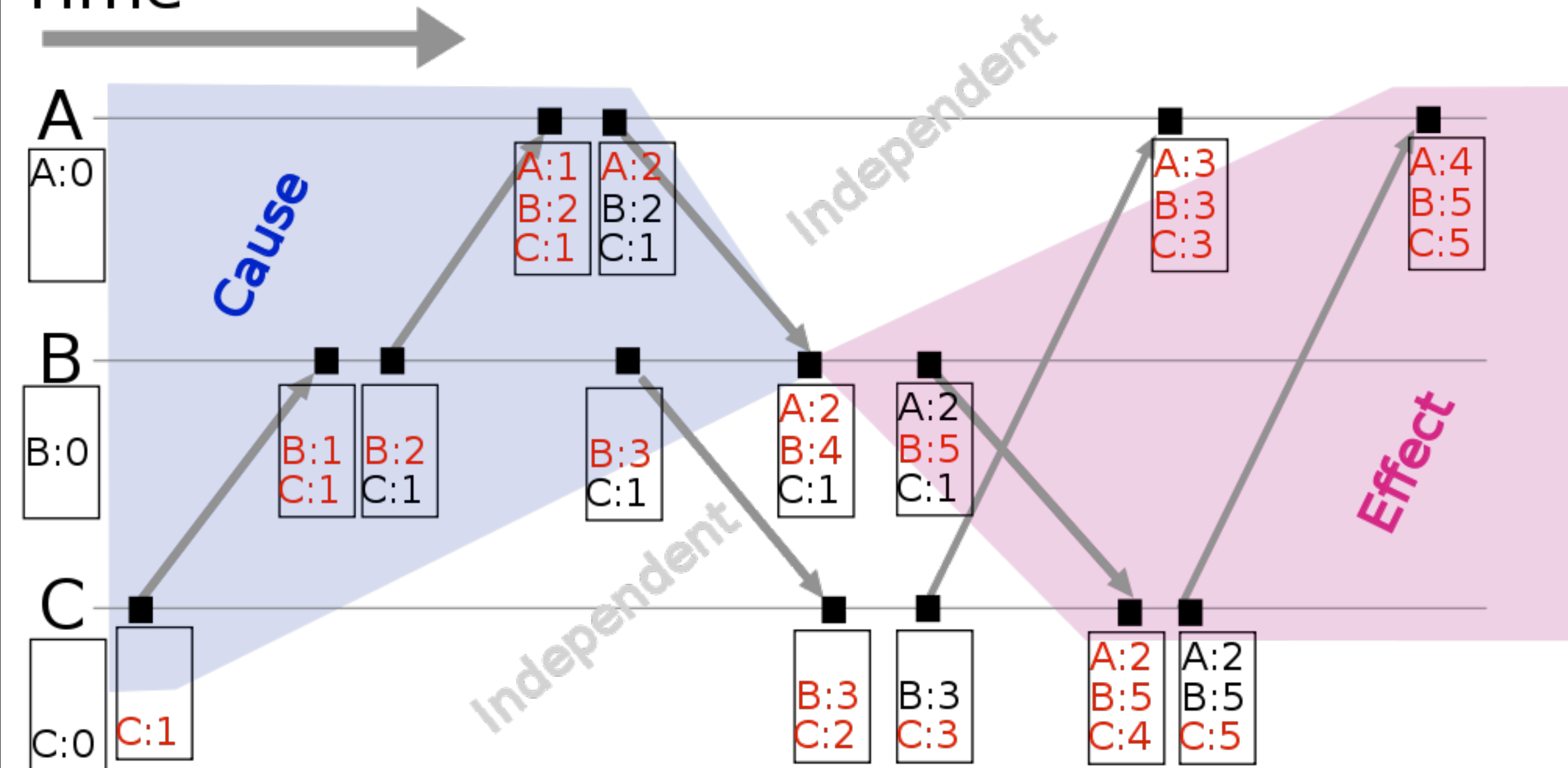
- A process increments its counter before each event in that process;
- When a process sends a message, it includes its counter value with the message;
- On receiving a message, the receiver process sets its counter to be the maximum of the message counter and its own counter incremented, before it considers the message received.

Lamport Clock Example

- Count on process1:
 - A1->A2 -> [write x A2]->A3 [notify x written A3]
- Count on process2:
 - B1->B2 -> [receive notify x written A3] ->B3 -> [read x B3]
- Count on process3:
 - C1 ->C2 -> [receive read x B3] -> C3 -> [write x **A2** - **C4**]

Vector Clocks

Time



Basic Clustering with JGroups

```
System.setProperty("jgroups.bind_addr", "127.0.0.1");  
System.setProperty("java.net.preferIPv4Stack", "true");  
System.setProperty("jgroups.udp.mcast_addr", "224.0.0.0");
```

```
JChannel channel = new JChannel();  
channel.connect("ServiceNodesCluster");  
channel.setReceiver(this);  
Message m = new Message(dest, src, "Data");  
channel.send(m);
```

```
-----  
public void receive(Message msg) {  
    Object data = msg.getObject();  
}  
-----
```

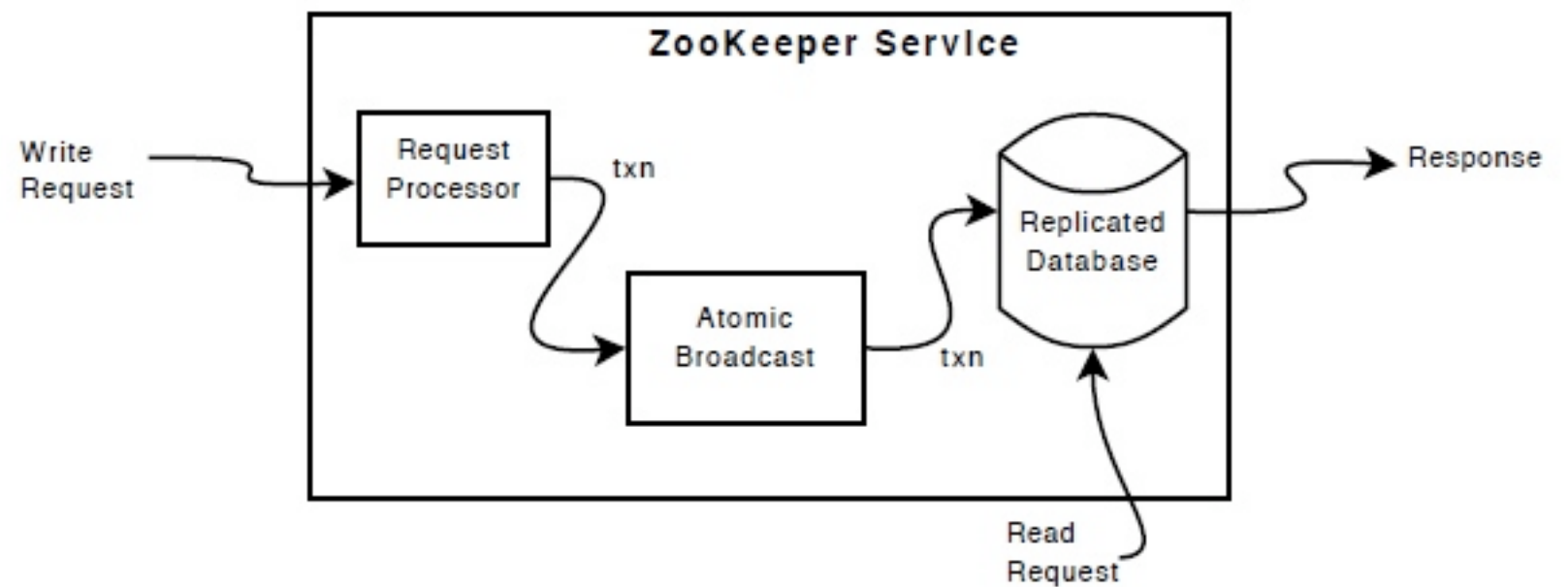
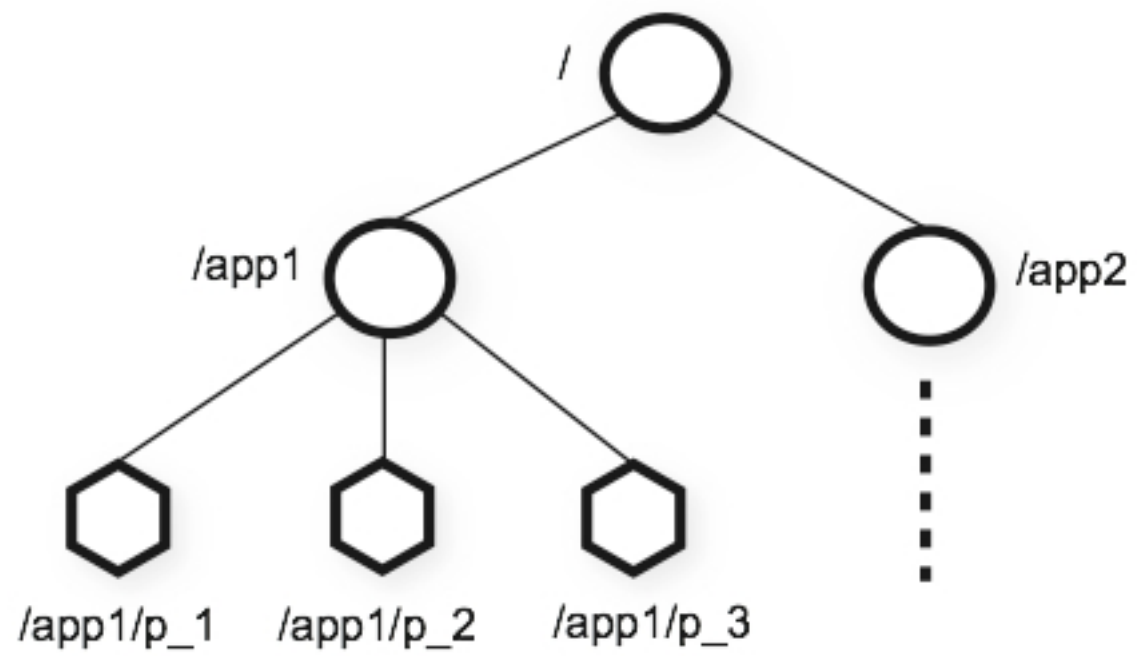
On a mac, enable multi casting:
`sudo route add -net 224.0.0.0 -interface lo0`

Creating a leader election

- Lets try with JGroups
 - Send a leader ping
 - If no one responds, become the leader
 - Respond to other's leader pings

ZooKeeper

- ZooKeeper is a hierarchical data store that
 - replicated for availability
 - replicated over a set of hosts called an ensemble
 - The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction log and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.
 - Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.
 - Guarantees order of updates
 - ZooKeeper stamps each update with a number that reflects the order of all ZooKeeper transactions. Subsequent operations can use the order to implement higher-level abstractions, such as synchronization primitives.
 - High performance memory storage
 - It is especially fast in "read-dominant" workloads. ZooKeeper applications run on thousands of machines, and it performs best where reads are more common than writes, at ratios of around 10:1.



Setting up zookeeper

- Download and unzip
- Edit data/myid file to setup an id for each cluster instance. Has to be a number 1-255
- Open up conf/zoo.cfg
 - Setup client port: clientPort=2181
 - Setup all servers in cluster: (The two ports are follower connection port: leader election port)

server.1=localhost:2887:3887

server.2=localhost:2888:3888

server.3=localhost:2889:3889

Basic zookeeper API

- Connect

```
zk = new ZooKeeper("localhost:2181", 3000,  
    watchedEvent -> {});
```

- Store

```
zk.create("/account", "Some account data".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE,  
CreateMode.EPHEMERAL);
```

- Check

```
if (zk.exists("/account", watchtrue/false) == null)
```

- Get

```
Stat stats = new Stat();  
byte[] data = zk.getData("/somepath", false, stats);
```

Storage classes

- EPHERMAL
 - Znode deleted upon disconnect
- EPHERMAL_SEQUENTIAL
 - ZNode is appended with a sequence and removed upon disconnect
- PERSISTENT
- PERSISTENT_SEQUENTIAL

Basic Zookeeper Interaction

```
ZooKeeper zk = new ZooKeeper("localhost:2181,localhost:2182", 10000,watchedEvent -> {
    System.out.println("Path: "+watchedEvent.getPath());
    System.out.println("Event "+watchedEvent);
});
zk.create("/somepath", "Hello There".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL, new
StringCallback() {
    public void processResult(int rc, String path, Object ctx, String name) {
        System.out.println("RC: "+rc);
    }
}, null);
Stat stats = new Stat();
byte[] data = zk.getData("/somepath", false, stats);
System.out.println("Data = "+new String(data));
zk.close();
```

Lets try

- Leader election with jgroups
- Leader election with zookeeper
- With multiple z-nodes

Zookeeper for configuration

- Zookeeper can be used as a good cluster configuration store and people do it with a lot of problems
- Zookeeper is linearizable - this should be a stronger application for zookeeper than as a configuration store.

Problems of Zookeeper as a service registry

- In CAP terms, ZooKeeper is CP,
 - It's consistent in the face of partitions, not available.
 - Since ZooKeeper is first and foremost a coordination service, having an eventually consistent design (being AP) would be bad.
 - Its core consensus algorithm, Zab, is therefore all about consistency.

Service Discovery

- For service discovery it's better to have information that may contain stale data than to have no data at all.
- It is much better to know what servers were available for a given service five minutes ago than to have no idea what things looked like due to a transient network partition
- Checkout netflix Eureka as a service discovery and configuration store

Claims debunked!

- Kyle Kingsbury does distributed database consistency analysis for a living. He created a tool called jepson.
- Checkout <http://jepson.io/>
- And his blogs at <https://aphyr.com/>

CAP - Partition is a given

- Even a single packet delivery loss is a partition.
 - CA systems are very difficult and can lead to huge performance delays
- Think AP and CP systems