

Goal of Spring Cloud

- Provide libraries to apply common patterns needed in distributed applications
 - Distributed / Versioned / Centralized Configuration Management
 - Service Registration and Discovery
 - Load Balancing
 - Service-to-service Calls
 - Circuit Breakers
 - Routing
 - ...

Spring Cloud Origins

- First, there was the Spring Framework (2004)
 - Alternative to low-level JEE approaches
- Next, Spring sub-projects emerged (2006 - present)
 - Spring Security, Web Flow, Integration, Batch, Web Services, XD, Social, Data, Boot, Session, etc.
 - Organized under Spring IO umbrella:



Where Does NETFLIX Fit Into All of This?

- Netflix reinvented itself since early 2007
 - Moved from DVD mailing to video-on-demand
 - Once USPS largest first-class customer
 - Now biggest source of North American Internet traffic in evenings.
 - Became Trailblazers in Cloud Computing
 - All Running on Amazon Web Services
 - Chose to publish many general-use technologies as Open-Source projects
 - Proprietary video-streaming technologies are still secret.

Capabilities

Measure	Eureka	Consul.io
Spring Cloud integration	Yes - Angel SR3	Yes – Brixton
Management in Container	I believe so. There are Docker targets. However, based on documentation, I'm not clear on configuration of peers.	Yes. The API implies Docker support.
Health monitoring of Endpoints	Yes. Removal of failed after 90 minutes (configurable)	Yes. Removal after 72 hours. (Fixed)
Multiple Datacenter	Possible	Yes per application documentation.
DNS support	No. Would require another package.	Yes. Built in.
HTTPS	Possible but not used in Netflix	Yes. Keygen and other configuration done via Consul CLI
Endpoint Update method	Delta content with occasional full content	Always full content
Recommended Number per datacenter	2 or more instances of Server connected as Peers	3-5 instances per Datacenter with ability to connect Datacenters.
Circuit Breaker	Supported with Hystrix and code modification	Not available
Health Check data	Hystrix with HystrixDashboard.	Built in to the Consul.io

collection	Aggregation via Turbine.	web access page.
Polyglot support	Requires addition of Sidecar.	Built in support.

¹

Spring and **NETFLIX**

- The Spring Team has always been forward looking
 - Trying to Focus on Applications of Tomorrow
 - Netflix OSS Mature and Battle-Tested; Why Reinvent?
 - Netflix OSS Not Necessarily Easy and Convenient
 - Spring Cloud provides easy interaction
 - Dependencies
 - Annotations

Required Dependencies

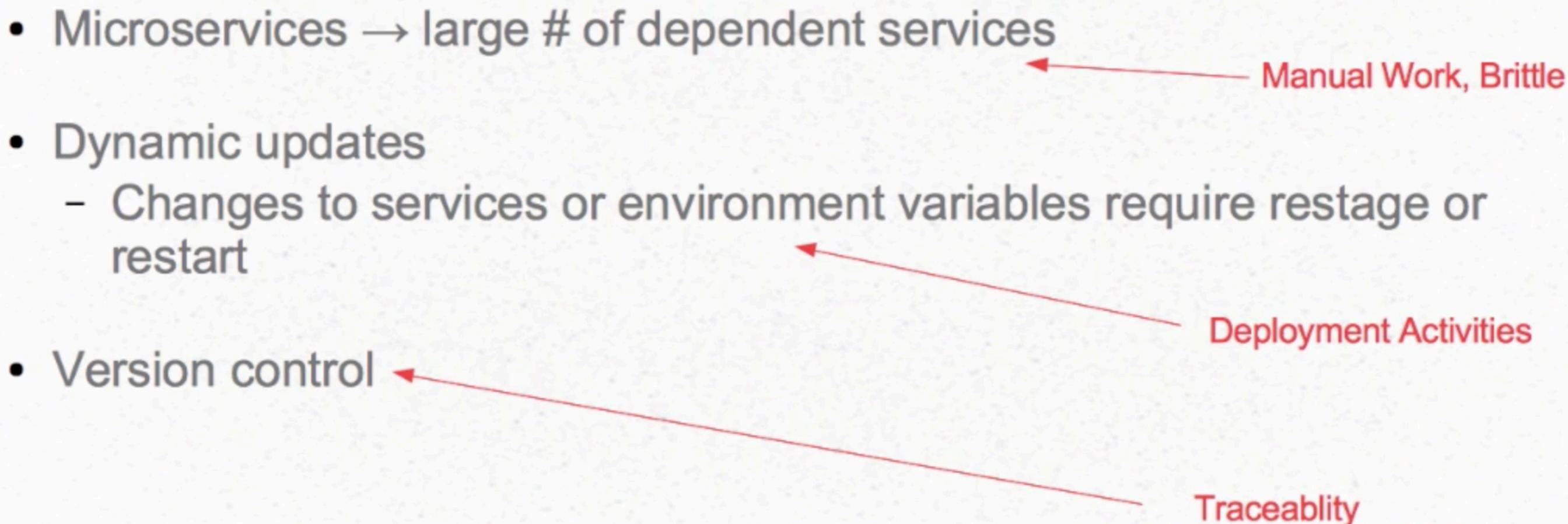
- Replace Spring Boot Parent
 - Spring Cloud projects are based on Spring Boot



```
<parent>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-parent</artifactId>
  <version>Angel.SR4</version>
</parent>

<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-...</artifactId>
</dependency>
```

Other Challenges

- Microservices → large # of dependent services
Manual Work, Brittle
- Dynamic updates
 - Changes to services or environment variables require restage or restartDeployment Activities
- Version controlTraceability

Desired Solution for Configuration

- Platform/Cloud-Independent solution
 - Language-independent too
- Centralized
 - Or a few discrete sources of our choosing
- Dynamic
 - Ability to update settings while an application is running
- Controllable
 - Same SCM choices we use with software
- Passive
 - Services (Applications) should do most of the work themselves by self-registering

Solution:

- Spring Cloud Config
 - Provides centralized, externalized, secured, easy-to-reach source of application configuration
- Spring Cloud Bus
 - Provides simple way to notify clients to config changes
- Spring Cloud Netflix Eureka
 - Service Discovery – Allows applications to register themselves as clients

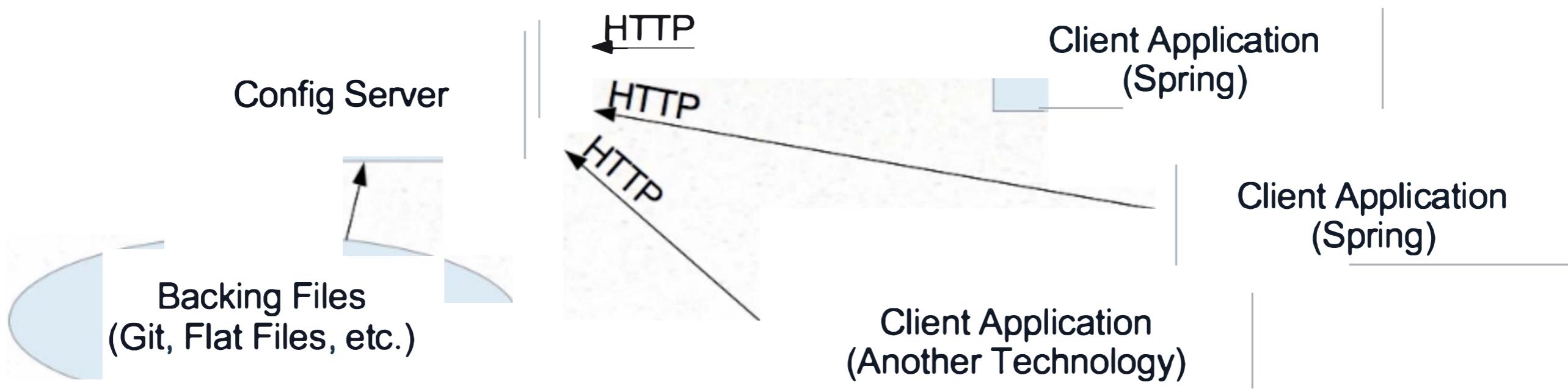
Spring Cloud Config

Designates a centralized server to serve-up configuration information

- Configuration itself can be backed by source control

Clients connect over HTTP and retrieve their configuration settings

- In addition to their own, internal sources of configuration



Spring Cloud Config Server

Include minimal dependencies in your POM (or Gradle)

- Spring Cloud Starter Parent
- Spring Cloud Config Server

```
<parent>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-parent</artifactId>
  <version>Angel.SR4 </version>
</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
  </dependency>
</dependencies>
```

Spring Cloud Config Server –

- application.yml – indicates location of configuration repository

```
spring:  
  cloud:  
    config:  
      server:  
        git:  
          uri: https://github.com/pradeepkrishnamurthy/  
          searchPaths: ConfigData
```

- ...or application.properties

Spring Cloud Config Server –

- Add `@EnableConfigServer`

```
@SpringBootApplication  
@EnableConfigServer  
public class Application {  
  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
  
}
```

EnvironmentRepository - Choices

- Spring Cloud Config Server uses an EnvironmentRepository
 - Two implementations available: Git and Native (local files)
- Implement EnvironmentRepository to use other sources.

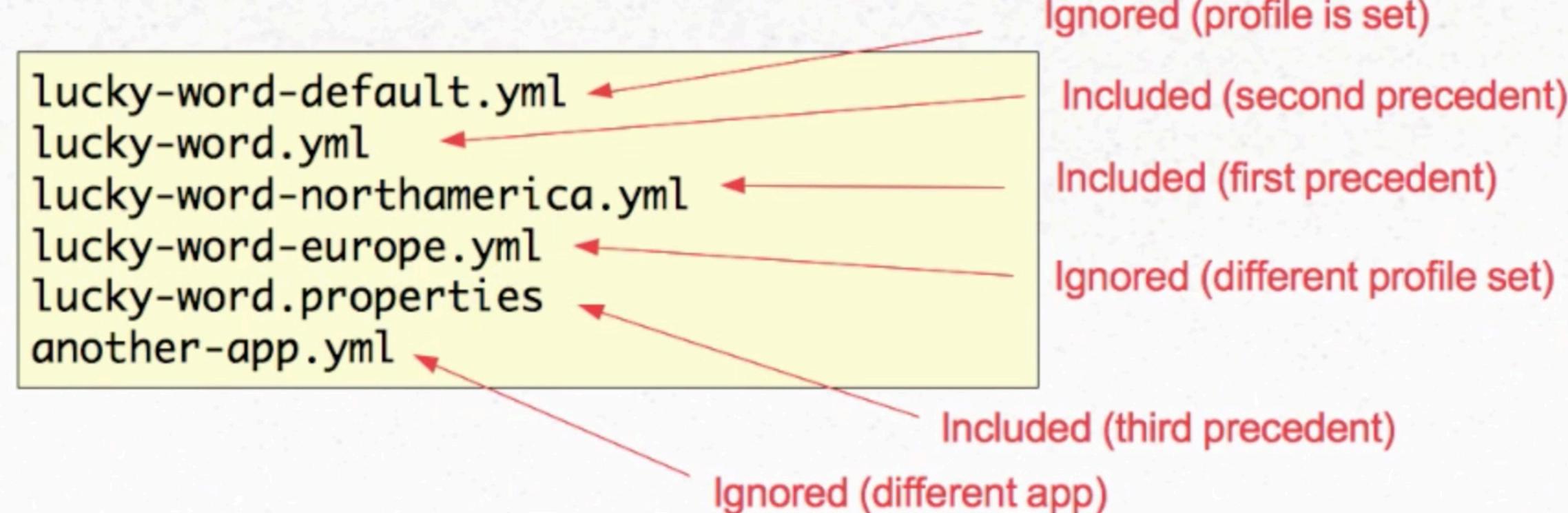
Environment Repository - Organization

- Configuration file naming convention:
 - <spring.application.name>-<profile>.yml
 - Or .properties (yml takes precedence)
 - spring.application.name – set by client application's bootstrap.yml (or .properties)
 - Profile – Client's spring.profiles.active
 - (set various ways)
- Obtain settings from server:
 - `http://<server>:<port>/<spring.application.name>/<profile>`
 - Spring clients do this automatically on startup

Environment Repository – Organization Example

Assume client application named “lucky-word” and profile set to “northamerica”

- Spring client (automatically) requests
 - /lucky-word/northamerica



.yml vs .properties

- Settings can be stored in either YAML or standard Java properties files
 - Both have advantages
 - Config server will favor .yml over .properties

```
# .properties file
spring.config.name=aaa
spring.config.location=bbb
spring.profiles.active=ccc
spring.profiles.include=ddd
```

```
# .yml file
---
spring:
  config:
    name: aaa
    location: bbb
  profiles:
    active: ccc
    include: ddd
```

Profiles

YAML Format can hold multiple profiles in a single file

```
# lucky-word-east.properties  
lucky-word: Clover
```

```
# lucky-word-west.properties  
lucky-word: Rabbit's Foot
```

```
# luckyword.yml  
---  
spring:  
  profiles: east  
  lucky-word: Clover  
  
---  
spring:  
  profiles: west  
  lucky-word: Rabbit's Foot
```

The Client Side

- How Properties work in Spring Applications
 - Spring apps have an Environment object
 - Environment object contains multiple PropertySources
 - Typically populated from environment variables, system properties, JNDI, developer-specified property files, etc.
 - Spring Cloud Config Client library simply adds another PropertySource
 - By connecting to server over HTTP
 - `http://<server>:<port>/<spring.application.name>/<profile>`
 - Result: Properties described by server become part of client application's environment

What if the Config Server is Down?

- Spring Cloud Config Server should typically run on several instances
 - So downtime should be a non-issue
- Client application can control policy of how to handle missing config server
 - `spring.cloud.config.failFast=true`
 - Default is false
- Config Server settings override local settings
 - Strategy: provide local fallback settings.

Service Discovery - Analogy

- When you sign into a chat client, what happens?
 - Client 'registers' itself with the server – server knows you are online.
 - The server provides you with a list of all the other known clients
- In essence, your client has “discovered” the other clients
 - ...and has itself been “discovered” by others

Search people..
Anson Hoy
Eri St. Martin
Hector Virn
Joe Sanantonio
Kevin Croker
Lisa Fernandez
Wei Teh
Adviti a Ban a
Axel Ulrich

Service Discovery

- Microservice architectures result in large numbers of inter-service calls
 - Very challenging to configure
- How can one application easily find all of the other runtime dependencies?
 - Manual configuration – Impractical, brittle
- Service Discovery provides a single 'lookup' service.
 - Clients register themselves, discover other registrants.
 - Solutions: Eureka, Consul, Etcd, Zookeeper, SmartStack, etc.

Eureka – Service Discovery Server and Client

- Part of Spring Cloud Netflix
 - Battle tested by Netflix
- Eureka provides a 'lookup' server.
 - Generally made highly available by running multiple copies
 - Copies replicate state of registered services.
- “Client” Services register with Eureka
 - Provide metadata on host, port, health indicator URL, etc.
- Client Services send heartbeats to Eureka
 - Eureka removes services without heartbeats.

Making a Eureka Server

- Just a regular Spring Boot web application with dependencies and `@EnableEurekaServer`:

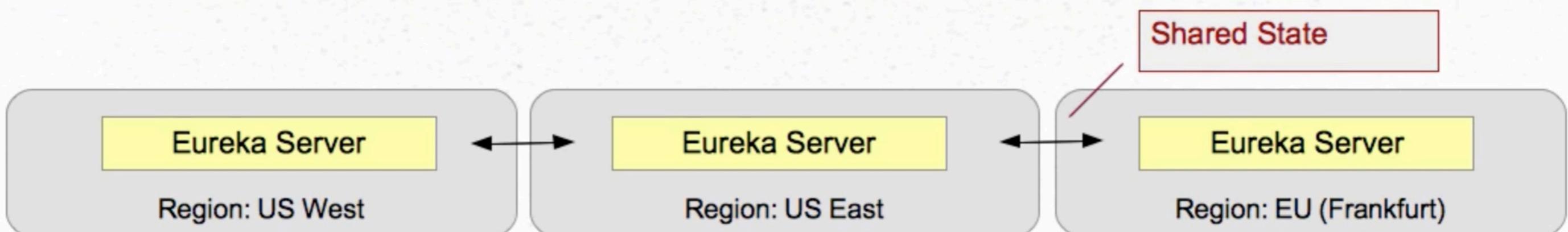
```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

```
@SpringBootApplication
@EnableEurekaServer
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

Multiple Servers

- Typically, multiple Eureka servers should be run simultaneously
 - Otherwise, you'll get many warnings in the log
 - Eureka servers communicate with each other to share state
 - Provides High Availability
- Each server should know URL to the others
 - Can be provided by Config Server
 - One server (JAR), multiple profiles



Multiple Servers

Configuration

- Common Configuration Options for Eureka Server:

- See <https://github.com/Netflix/eureka/wiki/Configuring-Eureka> for full list.

Control http port (any boot application)

```
server:  
  port: 8011  
eureka:  
  instance:  
    statusPageUrlPath: ${management.contextPath}/info  
    healthCheckUrlPath: ${management.contextPath}/health  
    hostname: localhost  
  client:  
    registerWithEureka: false  
    fetchRegistry: false  
    serviceUrl:  
      defaultZone: http://server:port/eureka/,http://server:port/eureka/
```

Comma separated list

Registering with Eureka

- From a Spring Boot application

- Add Dependency:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-eureka-server</artifactId>
</dependency>
```

- Only the location of Eureka itself requires explicit configuration.

```
@SpringBootApplication
@EnableDiscoveryClient
public class Application {
}
```

```
# application.properties
eureka.client.serviceUrl.defaultZone: http://server:8761/eureka/
```

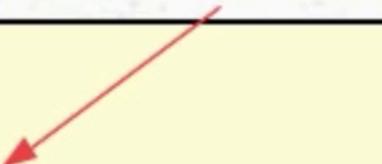
Default fallback.
Any Eureka instance will do
(we usually want several comma-separated URLs)

@EnableDiscoveryClient

- Automatically registers client with Eureka server
 - Registers the application name, host, and port
 - Using values from the Spring Environment.
 - But can be overridden.
 - Give your application a `spring.application.name`
- Makes this app an “instance” and a “client”
 - It can locate other services

Service ID (Eureka VIP)
Corresponds to
`spring.application.name`

```
@Autowired DiscoveryClient client;
public URI storeServiceUrl() {
    List<ServiceInstance> list = client.getInstances("STORES");
    if (list != null && list.size() > 0 ) {
        return list.get(0).getUri();
    }
    return null;
}
```



What is a Zone?

- Eureka server designed for multi-instance use
 - Standalone mode will actually warn you when it runs without any peers!
- Eureka Server does not persist service registrations
 - Relies on client registrations; always up to date, always in memory
 - Stateful application.
- Typical production usage – many Eureka server instances running in different availability zones / regions
 - Connected to each other as “peers”

Which Comes First? Eureka or Config Server?

Config First Bootstrap (default) – Use Config Server to configure location of Eureka server

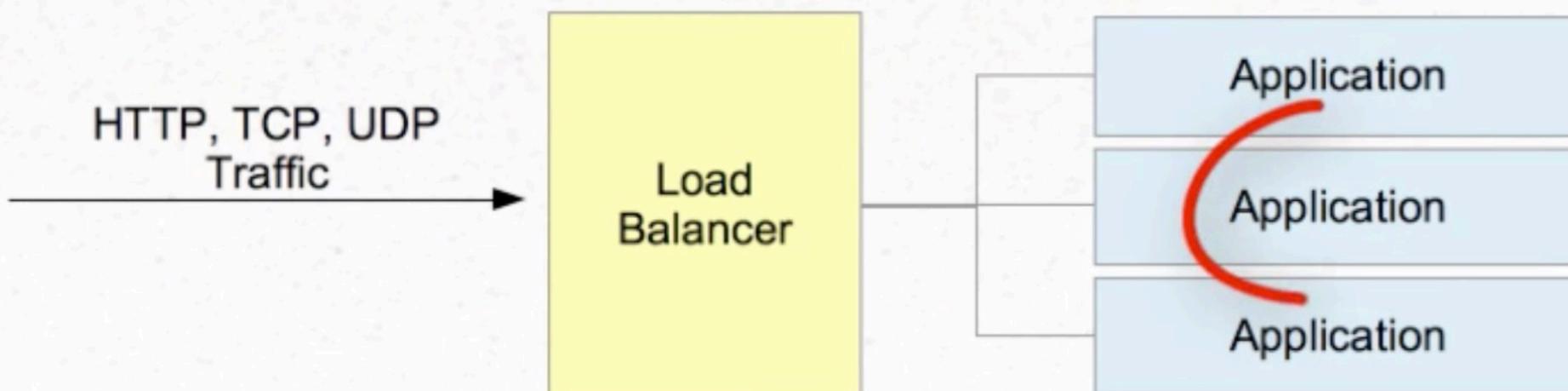
- Implies `spring.cloud.config.uri` configured in each app

Eureka First Bootstrap – Use Eureka to expose location to config server

- Config server is just another client
- Implies `spring.cloud.config.discovery.enabled=true` and `eureka.client.serviceUrl.defaultZone` configured in each app.
- Client makes two network trips to obtain configuration.

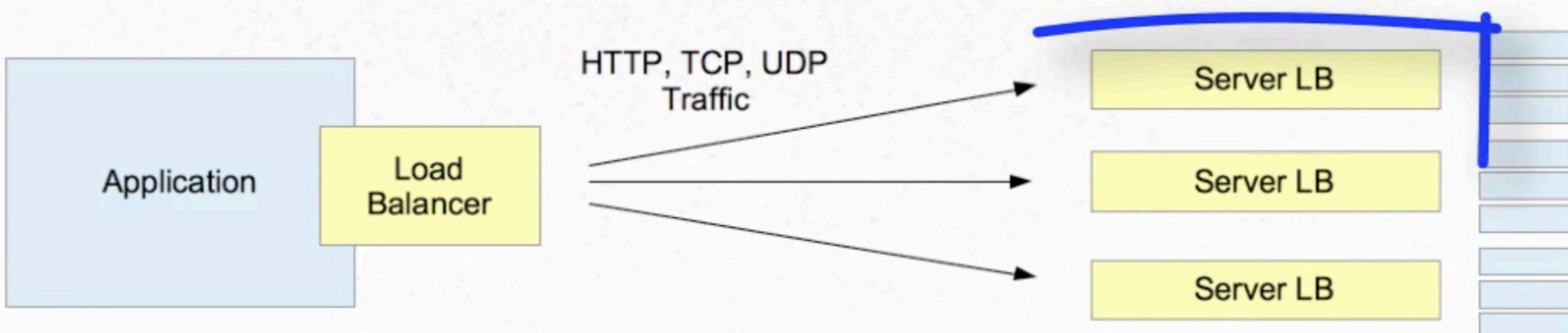
What is a Load Balancer?

- Traditional load balancers are server-side components
 - ▶ • Distribute incoming traffic among several servers
 - Software (Apache, Nginx, HA Proxy) or Hardware (F5, NSX, BigIP)



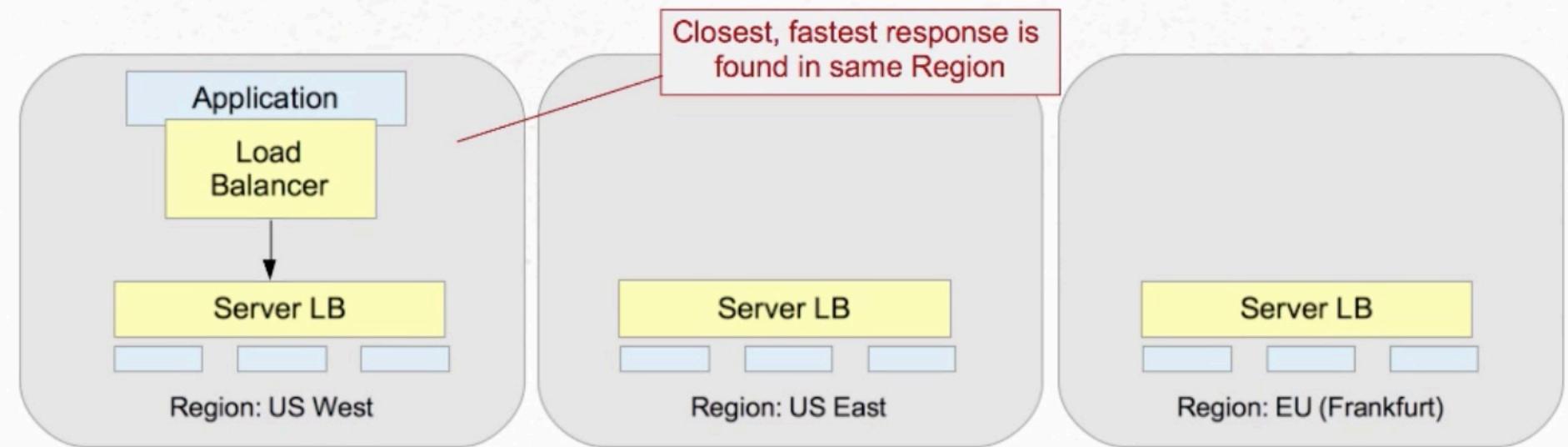
Client-Side Load Balancer

- Client-Side Load Balancer selects which server to call
 - Based on some criteria
 - Part of client software
 - Server can still employ its own load balancer



Why?

- Not all servers are the same
 - Some may be unavailable (faults)
 - Some may be slower than others (performance)
 - Some may be further away than others (regions)



Spring Cloud Netflix Ribbon

- Ribbon – Another part of the Netflix OSS family
 - Client side load balancer
 - Automatically integrates with service discovery (Eureka)
 - Built in failure resiliency (Hystrix)
 - Caching / Batching
 - Multiple protocols (HTTP, TCP, UDP)
- Spring Cloud provides an easy API Wrapper for using Ribbon.

List of Servers

- Determines what the list of possible servers are (for a given service (client))
 - Static – Populated via configuration
 - Dynamic – Populated via Service Discovery (Eureka)
- Spring Cloud default – Use Eureka when present on the classpath.

Example of “Static” server lists

application.yml

“stores” and “products” -
Examples of client-ids

```
stores:  
  ribbon:  
    listOfServers: store1.com,store2.com  
products:  
  ribbon:  
    listOfServers: productServer1.com, productServer2.com
```