Test creaeting pdf from markdown

Service Discovery

Session Overview

DURATION	LEVEL	PREREQUISITES
60 min	Intermediate	Basic microservices concepts, networking fundamentals

Learning Objectives

By the end of this session, you will be able to: - Understand the need for service discovery in distributed systems - Compare client-side vs server-side discovery patterns - Evaluate different service registry technologies (Consul, etcd, ZooKeeper) - Implement health checking and failure detection - Design DNS-based service discovery solutions

Agenda

TIME	TOPIC
0-10 min	Why Service Discovery?
10-25 min	Discovery Patterns
25-40 min	Service Registry Technologies
40-50 min	Health Checking & DNS-Based Discovery
50-60 min	Practical Exercise & Q&A

1. Why Service Discovery?

The Problem

In traditional monolithic applications, service locations are typically static and configured at deployment time. In microservices architectures, this approach breaks down:

```
flowchart TB
    subgraph Traditional["Traditional (Static)"]
        TC[Client] -->|"config.properties<br/>server=192.168.1.10"| TS[Server<br/>192.168.1.10]
    end

    subgraph Microservices["Microservices (Dynamic)"]
        SA[Service A] -->|"Where are the<br/>order services?"| Orders
        subgraph Orders["Order Services (scaling)"]
            O1[Order-1]
            O2[Order-2]
            O3[Order-3]
            O4[Order-4<br/>🆕 new]
        end
    end
```

Challenges in Dynamic Environments

1. **Auto-scaling**: Instances are added/removed based on load
2. **Failures**: Instances crash and restart on different hosts
3. **Deployments**: Rolling updates change instance locations
4. **Containerization**: Containers get dynamic IPs

Service Discovery Components

```
flowchart TB
    subgraph SD["Service Discovery System"]
        SR["Service Registry<br/>• Register<br/>• Deregister<br/>• Store"]
        DC["Service Discovery Client<br/>• Query<br/>• Subscribe<br/>• Cache"]
        HC["Health Checking<br/>• Liveness<br/>• Readiness<br/>• Monitor"]
    end
```

2. Discovery Patterns

Client-Side Discovery

The client is responsible for determining the network locations of available service instances and load balancing requests across them.

```
flowchart LR
    subgraph Client["Client"]
        DC["Discovery<br/>Client"]
        LB["Load<br/>Balancer"]
    end

    subgraph Registry["Service Registry"]
        SR["order-svc:<br/>• 10.0.1.5<br/>• 10.0.1.6<br/>• 10.0.1.7"]
    end

    SI[Service Instance]

    DC -->|"1. Query"| SR
    SR -->|"2. Return instances"| DC
    DC --> LB
    LB -->|"3. Direct request"| SI
```

Implementation Example (Java with Netflix Eureka):

```
@Configuration
@EnableDiscoveryClient
public class ServiceDiscoveryConfig {

    @Bean
    @LoadBalanced
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}

@Service
public class OrderServiceClient {

    @Autowired
    private RestTemplate restTemplate;

    @Autowired
    private DiscoveryClient discoveryClient;

    // Using service name instead of hardcoded URL
    public Order getOrder(String orderId) {
        return restTemplate.getForObject(
            "http://order-service/orders/" + orderId,
            Order.class
        );
    }

    // Manual discovery for more control
    public List<ServiceInstance> getOrderServiceInstances() {
        return discoveryClient.getInstances("order-service");
    }
}
```

Pros: - Client has full control over load balancing strategy - No single point of failure in the load balancer - Can implement sophisticated routing (e.g., zone-aware)

Cons: - Discovery logic coupled with client code - Must implement for each programming language - Clients must handle registry unavailability

Server-Side Discovery

A load balancer or router handles service discovery, and clients make requests to the load balancer.

```
flowchart LR
    Client[Client]

    subgraph LB["Load Balancer"]
        DC["Discovery<br/>Client"]
        Route["Route<br/>Request"]
    end

    subgraph Registry["Service Registry"]
        SR["order-svc:<br/>• 10.0.1.5<br/>• 10.0.1.6"]
    end

    SI[Service Instance]

    Client -->|"1. Request"| DC
    DC -->|"2. Query"| SR
    SR -->|"3. Return"| DC
    DC --> Route
    Route -->|"4. Forward"| SI
    SI -->|"5. Response"| Client
```

Implementation Example (Kubernetes Service):

```yaml
# Kubernetes Service acts as server-side discovery
apiVersion: v1
kind: Service
metadata:
  name: order-service
spec:
  selector:
    app: order
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP

---
# Client just uses the service name
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-gateway
spec:
  template:
    spec:
      containers:
        - name: gateway
          env:
            - name: ORDER_SERVICE_URL
              value: "http://order-service"  # DNS resolves to service
```

Pros: - Simpler client implementation - Language-agnostic - Centralized load balancing policies

Cons: - Load balancer can become a bottleneck - Additional network hop - Must ensure load balancer high availability

Pattern Comparison

ASPECT	CLIENT-SIDE	SERVER-SIDE
Client Complexity	High	Low
Network Hops	Fewer	More
Load Balancer	Distributed	Centralized
Language Support	Per-language	Universal
Flexibility	High	Medium
Examples	Netflix Ribbon, gRPC	Kubernetes, AWS ALB

3. Service Registry Technologies

Consul

HashiCorp Consul provides service discovery with built-in health checking, KV store, and multi-datacenter support.

```
flowchart TB
    subgraph DC1["Datacenter 1"]
        CS1["Consul Server<br/>(Leader)"]
        CS1a["Server"]
        CS1b["Server"]
        CA1["Agent (svc)"]
        CA2["Agent (svc)"]

        CS1 --- CS1a
        CS1 --- CS1b
        CS1a --- CA1
        CS1b --- CA2
    end

    subgraph DC2["Datacenter 2"]
        CS2["Consul Server<br/>(Leader)"]
        CS2a["Server"]
        CS2b["Server"]
        CA3["Agent (svc)"]
        CA4["Agent (svc)"]

        CS2 --- CS2a
        CS2 --- CS2b
        CS2a --- CA3
        CS2b --- CA4
    end

    CS1 <-->|"WAN Gossip"| CS2
```

Service Registration:

```
// service.json
{
  "service": {
    "name": "order-service",
    "id": "order-service-1",
    "port": 8080,
    "tags": ["v1", "primary"],
    "check": {
      "http": "http://localhost:8080/health",
      "interval": "10s",
      "timeout": "5s"
    }
  }
}
```

```
# Register service
consul services register service.json

# Query service
consul catalog services
consul catalog nodes -service=order-service

# DNS query
dig @127.0.0.1 -p 8600 order-service.service.consul
```

Go Client Example:

```go
package main

import (
    "fmt"
    "github.com/hashicorp/consul/api"
)

func main() {
    // Create client
    config := api.DefaultConfig()
    client, _ := api.NewClient(config)

    // Register service
    registration := &api.AgentServiceRegistration{
        ID:      "order-service-1",
        Name:    "order-service",
        Port:    8080,
        Check: &api.AgentServiceCheck{
            HTTP:     "http://localhost:8080/health",
            Interval: "10s",
        },
    }
    client.Agent().ServiceRegister(registration)

    // Discover services
    services, _, _ := client.Health().Service("order-service", "", true, nil)
    for _, svc := range services {
        fmt.Printf("Found: %s:%d\n", svc.Service.Address, svc.Service.Port)
    }
}
```

etcd

etcd is a distributed key-value store that provides a reliable way to store data across a cluster. It’s the backbone of Kubernetes service discovery.

```
flowchart TB
    subgraph Cluster["etcd Cluster"]
        N1["Node 1<br/>(Leader)"]
        N2["Node 2<br/>(Follower)"]
        N3["Node 3<br/>(Follower)"]

        N1 <-->|"Raft Consensus"| N2
        N2 <-->|"Raft Consensus"| N3
        N3 <-->|"Raft Consensus"| N1
    end

    SA[Service A]
    SB[Service B]
    SC[Service C]

    Cluster --> SA
    Cluster --> SB
    Cluster --> SC
```

Service Registration with etcd:

```python
import etcd3
import json
import time
from threading import Thread

class ServiceRegistry:
    def __init__(self, etcd_host='localhost', etcd_port=2379):
        self.client = etcd3.client(host=etcd_host, port=etcd_port)
        self.lease = None

    def register(self, service_name, instance_id, host, port, ttl=30):
        """Register service with TTL-based lease"""
        key = f"/services/{service_name}/{instance_id}"
        value = json.dumps({
            "host": host,
            "port": port,
            "registered_at": time.time()
        })

        # Create lease for automatic deregistration
        self.lease = self.client.lease(ttl)
        self.client.put(key, value, lease=self.lease)

        # Start heartbeat thread
        Thread(target=self._heartbeat, daemon=True).start()

    def _heartbeat(self):
        """Keep lease alive"""
        while True:
            self.lease.refresh()
            time.sleep(10)

    def discover(self, service_name):
        """Get all instances of a service"""
        prefix = f"/services/{service_name}/"
        instances = []

        for value, metadata in self.client.get_prefix(prefix):
            instances.append(json.loads(value))

        return instances
    def watch(self, service_name, callback):
        """Watch for service changes"""
        prefix = f"/services/{service_name}/"
        events_iterator, cancel = self.client.watch_prefix(prefix)

        for event in events_iterator:
            callback(event)

# Usage
registry = ServiceRegistry()
registry.register("order-service", "instance-1", "10.0.1.5", 8080)
instances = registry.discover("order-service")
```

ZooKeeper

Apache ZooKeeper provides distributed coordination and is used by many systems for service discovery.

```
flowchart TB
    subgraph ZK["ZooKeeper Data Model"]
        Root["/"]
        Services["/services"]
        Config["/config"]

        OrderSvc["/order-service"]
        UserSvc["/user-service"]
        DB["/database"]

        O1["instance-001 (ephemeral)<br/>host:10.0.1.5, port:8080"]
        O2["instance-002 (ephemeral)<br/>host:10.0.1.6, port:8080"]
        O3["instance-003 (ephemeral)<br/>host:10.0.1.7, port:8080"]

        U1["instance-001 (ephemeral)"]
        U2["instance-002 (ephemeral)"]

        DBConfig["url: jdbc:mysql://..."]

        Root --> Services
        Root --> Config
        Services --> OrderSvc
        Services --> UserSvc
        Config --> DB
        OrderSvc --> O1
        OrderSvc --> O2
        OrderSvc --> O3
        UserSvc --> U1
        UserSvc --> U2
        DB --> DBConfig
    end
```

Note: Ephemeral nodes are automatically deleted when the session ends (service crashes)

Java Implementation with Curator:

```
        import org.apache.curator.framework.CuratorFramework;
        import org.apache.curator.framework.CuratorFrameworkFactory;
        import org.apache.curator.retry.ExponentialBackoffRetry;
        import org.apache.curator.x.discovery.*;

        public class ZookeeperServiceDiscovery {

            private final ServiceDiscovery<ServiceInstance> serviceDiscovery;
            private final CuratorFramework client;

            public ZookeeperServiceDiscovery(String zkConnection) throws Exception {
                client = CuratorFrameworkFactory.newClient(
                    zkConnection,
                    new ExponentialBackoffRetry(1000, 3)
                );
                client.start();

                serviceDiscovery =
ServiceDiscoveryBuilder.builder(ServiceInstance.class)
                    .client(client)
                    .basePath("/services")
                    .build();
                serviceDiscovery.start();
            }

            public void register(String serviceName, String host, int port) throws
Exception {
                ServiceInstance<ServiceInstance> instance =
ServiceInstance.<ServiceInstance>builder()
                    .name(serviceName)
                    .address(host)
                    .port(port)
                    .build();

                serviceDiscovery.registerService(instance);
            }

            public Collection<ServiceInstance<ServiceInstance>> discover(String
serviceName)
                    throws Exception {
                return serviceDiscovery.queryForInstances(serviceName);
            }
            // Watch for changes
            public void watchService(String serviceName, ServiceCacheListener listener)
                    throws Exception {
                ServiceCache<ServiceInstance> cache = serviceDiscovery
                    .serviceCacheBuilder()
                    .name(serviceName)
                    .build();

                cache.addListener(listener);
                cache.start();
            }
        }
```

Technology Comparison

FEATURE	CONSUL	ETCD	ZOOKEEPER
Consensus	Raft	Raft	ZAB
Health Checking	Built-in	External	External
Multi-DC	Native	Manual	Manual
DNS Interface	Yes	No	No
KV Store	Yes	Yes	Yes
ACL	Yes	Yes	Yes
Language	Go	Go	Java
Kubernetes	Supported	Native	Supported

4. Health Checking & DNS-Based Discovery

Health Check Types

```
flowchart LR
    subgraph Liveness["1. Liveness Check"]
        L_Q["Is the process running?"]
        LC[Checker] -->|"GET /health/live"| LS[Service]
        LS -->|"200 OK"| LC
    end

    subgraph Readiness["2. Readiness Check"]
        R_Q["Can it handle traffic?"]
        RC[Checker] -->|"GET /health/ready"| RS[Service]
        RS -->|"200 OK / 503"| RC
    end

    subgraph Startup["3. Startup Check"]
        S_Q["Has it finished initializing?"]
        SC[Checker] -->|"GET /health/startup"| SS[Service]
        SS -->|"200 OK / 503"| SC
    end
```

Health Check Implementation:

```java
@RestController
@RequestMapping("/health")
public class HealthController {

    @Autowired
    private DataSource dataSource;

    @Autowired
    private RedisTemplate<String, String> redisTemplate;

    @GetMapping("/live")
    public ResponseEntity<Map<String, String>> liveness() {
        // Simple check - is the process running?
        return ResponseEntity.ok(Map.of("status", "UP"));
    }

    @GetMapping("/ready")
    public ResponseEntity<Map<String, Object>> readiness() {
        Map<String, Object> health = new HashMap<>();
        boolean ready = true;

        // Check database
        try {
            dataSource.getConnection().isValid(5);
            health.put("database", "UP");
        } catch (Exception e) {
            health.put("database", "DOWN");
            ready = false;
        }

        // Check Redis
        try {
            redisTemplate.getConnectionFactory().getConnection().ping();
            health.put("redis", "UP");
        } catch (Exception e) {
            health.put("redis", "DOWN");
            ready = false;
        }

        health.put("status", ready ? "UP" : "DOWN");

        return ready
```

```
                    ? ResponseEntity.ok(health)
                    : ResponseEntity.status(503).body(health);
            }
        }
```

DNS-Based Service Discovery

DNS-based discovery uses standard DNS protocols to resolve service names to IP addresses.

```
flowchart LR
    Client[Client]
    DNS["DNS Server<br/>A Records:<br/>• 10.0.1.5<br/>• 10.0.1.6<br/>•
10.0.1.7"]
    Server["10.0.1.5:8080"]

    Client -->|"1. DNS Query<br/>order.svc.local"| DNS
    DNS -->|"2. IP Addresses"| Client
    Client -->|"3. Direct Connection"| Server
```

SRV Records (with port info):

```
_http._tcp.order.svc.local. 86400 IN SRV 0 5 8080 order1
_http._tcp.order.svc.local. 86400 IN SRV 0 5 8080 order2
```

Kubernetes DNS Discovery:

```yaml
# Service creates DNS entries automatically
apiVersion: v1
kind: Service
metadata:
  name: order-service
  namespace: production
spec:
  selector:
    app: order
  ports:
    - port: 80
      targetPort: 8080

# DNS names created:
# - order-service.production.svc.cluster.local (full)
# - order-service.production (within cluster)
# - order-service (within same namespace)
```

```
// Go client using DNS
package main

import (
    "net"
    "net/http"
)

func main() {
    // Simple DNS resolution
    ips, _ := net.LookupHost("order-service.production.svc.cluster.local")

    // SRV record lookup for port info
    _, addrs, _ := net.LookupSRV("http", "tcp", "order-service.production.svc.cluster.local")

    for _, addr := range addrs {
        fmt.Printf("Target: %s, Port: %d\n", addr.Target, addr.Port)
    }

    // HTTP client with DNS
    client := &http.Client{}
    resp, _ := client.Get("http://order-service/api/orders")
}
```

5. Implementation Patterns

Self-Registration Pattern

Services register themselves with the registry on startup.

```
sequenceDiagram
    participant S as Service Instance
    participant R as Registry

    Note over S,R: Service Startup
    S->>R: 1. Register
    loop Periodic
        S->>R: 2. Heartbeat
    end

    Note over S,R: Service Shutdown
    S->>R: 3. Deregister
```

```
        @Component
        public class ServiceRegistration implements
ApplicationListener<ApplicationReadyEvent> {

            @Autowired
            private ConsulClient consulClient;

            @Value("${spring.application.name}")
            private String serviceName;

            @Value("${server.port}")
            private int port;

            private String instanceId;

            @Override
            public void onApplicationEvent(ApplicationReadyEvent event) {
                instanceId = serviceName + "-" + UUID.randomUUID().toString();

                NewService service = new NewService();
                service.setId(instanceId);
                service.setName(serviceName);
                service.setPort(port);
                service.setCheck(createHealthCheck());

                consulClient.agentServiceRegister(service);
            }

            @PreDestroy
            public void deregister() {
                consulClient.agentServiceDeregister(instanceId);
            }

            private NewService.Check createHealthCheck() {
                NewService.Check check = new NewService.Check();
                check.setHttp("http://localhost:" + port + "/health");
                check.setInterval("10s");
                return check;
            }
        }
```

Third-Party Registration Pattern

A separate registrar service handles registration on behalf of services.

```
flowchart LR
    S[Service Instance]
    R["Registrar<br/>(Sidecar)"]
    Reg[Registry]

    R -->|"1. Monitor health"| S
    R -->|"2. Register"| Reg
    R -->|"3. Heartbeat"| Reg
```

Examples: Kubernetes, Netflix Prana, Registrator

Key Takeaways

1. **Service discovery is essential** for dynamic microservices environments where instances come and go
2. **Two main patterns**: Client-side (more control, more complexity) vs Server-side (simpler clients, centralized)
3. **Choose the right registry**: Consul for multi-DC, etcd for Kubernetes, ZooKeeper for existing Hadoop ecosystems
4. **Health checking is critical**: Implement liveness, readiness, and startup probes appropriately
5. **DNS-based discovery** provides simplicity and language-agnostic access but has caching challenges
6. **Consider failure modes**: What happens when the registry is unavailable? Cache locally and fail gracefully

Practical Exercise

Design Challenge: Multi-Region Service Discovery

Design a service discovery solution for a global e-commerce platform with the following requirements:

1. **Services deployed across 3 regions**: US-East, EU-West, Asia-Pacific
2. **Latency-sensitive routing**: Prefer local region, fallback to others
3. **Graceful degradation**: Continue operating if one region’ s registry fails
4. **Health-aware routing**: Only route to healthy instances

Consider: - Which discovery pattern (client-side vs server-side)? - Which registry technology? - How to handle cross-region discovery? - Caching strategy for registry data? - Failover behavior?

Deliverables: 1. Architecture diagram showing discovery flow 2. Sequence diagram for service registration 3. Failure scenario analysis

Q&A

Common questions to explore: - How does Kubernetes handle service discovery internally? - What’ s the difference between service mesh discovery and traditional discovery? - How do you migrate from static configuration to dynamic discovery? - What are the security considerations for service registries?