Service Discovery

MICROSERVICES

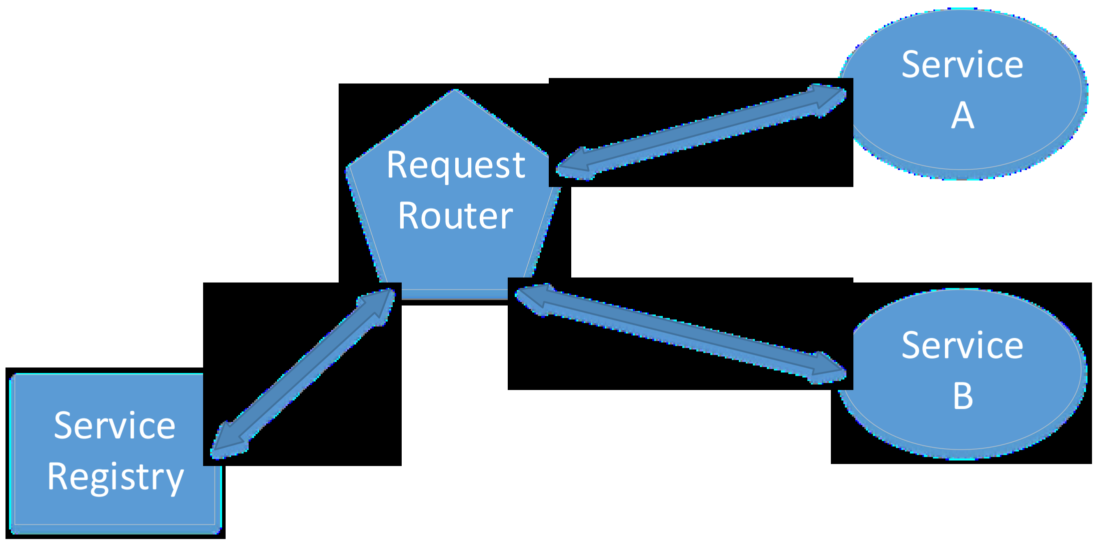
Content

- Underlying concepts
  - Client-side discovery
  - Server-side discovery
- DNS
- Key value stores (ZooKeeper, etcd)
- Specialized products (consul)

Introduction

- Service discovery can be used for many tasks:
  - Service resolution for cross service communication
  - Dynamic load balancing configuration
  - Dynamic monitoring configuration
  - …
- Approaches:
  - Static configuration files (forget about this…)
  - DNS
  - Dynamic solutions
- Patterns:
  - Client-side discovery
  - Server-side discovery
- Additionally a few products also include:
  - Configuration stores
  - Health checks (basically a kind of monitoring)
  - Events on changes (e.g. new services, changes in the configuration store…)

Server-side service discovery



Server-side service discovery

Context:

- Services are registered at a central service registry

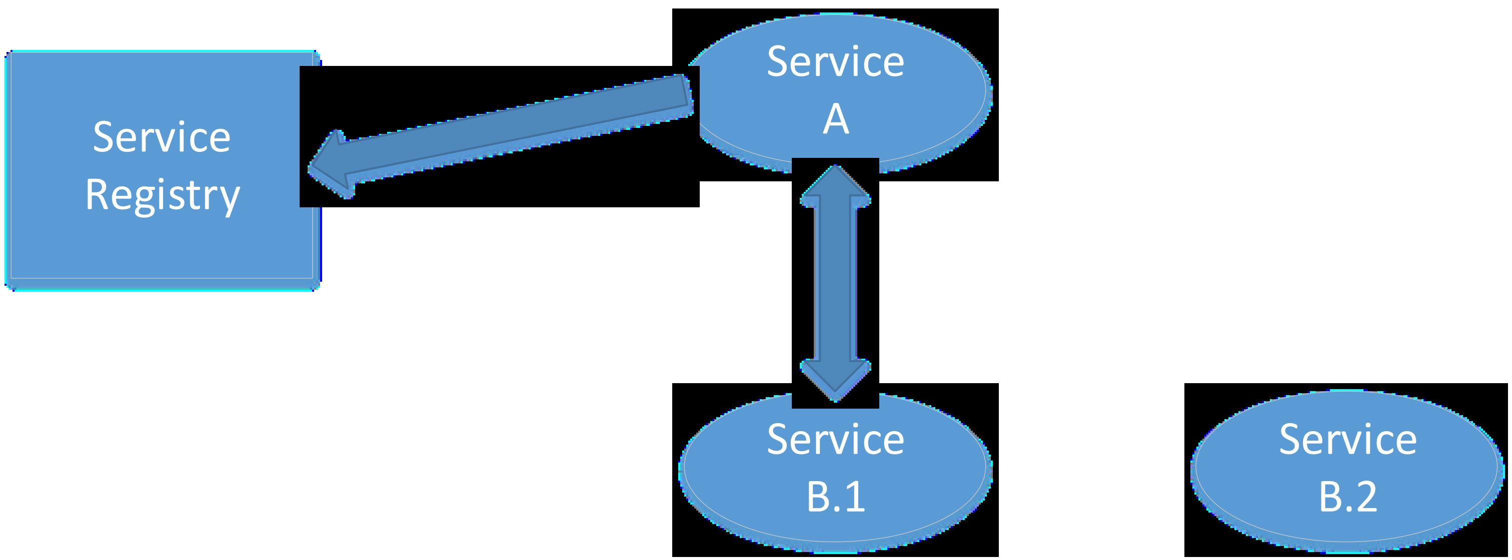
Problem:

- Service A wants to contact service B

Solution:

1. Service A contacts router
2. If router is not a service registry itself it contacts the service registry to get an address and port of service B
3. Router redirects request to instance of service B
4. Router redirects response to service A

Client-side service discovery



Client-side service discovery

Context:

- Services are registered at a central service registry

Problem:

- Service A wants to contact service B

Solution:

1. Service A asks the service registry for one or all known instances of service B (depends on the kind of service registry)
2. Service A uses response of the service registry to contact service B directly (assuming service A is aware of the API of service B)

Service registration

Self registration

- Every client/service registers himself at the service registry
- Every client has to deregister himself on failures or when quitting
- Every client has to deal with the API of the service registry himself
- E.g. Netflix Eureka

3rd party registration

- Clients/services are registered by a external instance
- Whenever a client exits the external component deregisters him
- The external component has to monitor every known service to ensure that it's still available
- E.g. registrator (Docker), Nomad

DNS – record types (selection)

Record name	Explanation
A or AAAA	Host entries (e.g. www.google.de – IPv4: 172.217.21.35 and IPv6: 2a00:1450:4016:80d::2003)
CNAME	Alias of a host entry (e.g. www.fh-rosenheim.de and fh-rosenheim.de)
SRV	Service location record (includes port of the service)
TXT	Often carries machine-readable data (often used e.g. for domain validation in Azure, C&C servers,…)
NAPTR	Name Authority Pointer – allows regular-expression-based rewriting of domain names (e.g. to form URIs)

https://en.wikipedia.org/wiki/List_of_DNS_record_types

DNS as service registry

- A (or AAAA) can be used to locate services (a single A record may contain multiple IP addresses e.g. amazon.com)
- SRV records are even better because SRV records also store the port of service
- Every instance has to register itself at a DNS server or a 3rd party service has to look for new instances and register them within a DNS server
- Developers and administrators are required to create a common schema for service naming

DNS – naming schemas

Schema example	Use case
<servicename>-<env>.domain.tld	All environments share the same domain/DNS server
<servicename>.<env>.domain.tld	Subdomain per environment (e.g. test.domain.tld and staging.domain.tld, keep prod on domain.tld)
<servicename>.env-domain.tld	Separate domains and DNS servers per environment

DNS as service registry

Pros:

- Very easy to implement
- No special software required
- Most stacks already support DNS queries

Cons:

- TTL of entries (stale entries)
- Many points where DNS caching happens, difficult to invalidate if services are ephemeral
- Dynamic registration difficult in self hosted environments (most DNS servers don't have REST APIs)

ZooKeeper

- One of the oldest tools which can be used for service discovery
- Developed as part of the Hadoop project
- Open-source – part of the Apache foundation
- Hierarchical key-value store
- Relies on running multiple nodes (recommended at least 3 nodes)
- Can also be used as configuration store
- Clients may subscribe to one or multiple keys to get notified when:
  - A new key was created
  - A key was deleted
  - A value was changed
  - …
- Also used for Kafka, Cassandra,…
- REST interface, client libraries available for many languages/ frameworks

https://zookeeper.apache.org/

Eureka

- Developed by Netflix for their own microservice architecture/platform as open-source project (Apache 2.0 license)
- Very targeted (in opposite to ZooKeeper and Consul)
- Provides basic load balancing capabilities (round-robin lookup for services)
- Java and REST API available
- Clients have to register themselves (Netflix uses Eureka in every component so this isn't a big deal for them); problematic in polyglot environments

https://github.com/Netflix/eureka

NETFLIX OSS

Consul

- Developed by Hashicorp (in Go) as open-source project (Mozilla open-source license)
- Can be used as configuration store
- Developed to serve as service discovery (unlike etcd or ZooKeeper)
- HTTP (RESTful) **and** DNS API (can be used as drop in replacement for an already existing DNS based solution)
- Built-in functionality for health checks (HTTP checks, TCP checks, check for running Docker containers or custom scripts) to detect unhealthy services and exclude them from the discovery
- Highly fault tolerant
- Based on a Gossip protocol (Serf)

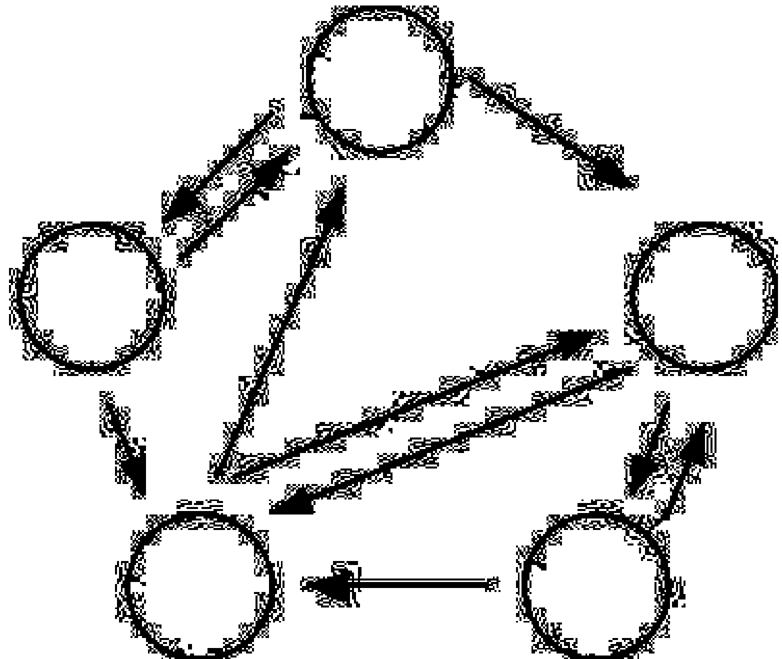
Gossip protocol

- In Consul used to manage membership and broadcast messages
- In Consul (Serf) based on Scalable Weakly-consistent Infection-style Process Group Membership Protocol (SWIM)
- A gossip protocol satisfies the following conditions:
  - Periodic, pairwise, inter-process (or network) interactions
  - The information exchanged during these interactions is of bounded size
  - Agents are synchronizing their state when they interact with each other
  - Reliable communication is **not** assumed
  - The frequency of the interactions is low compared to typical message latencies so that the protocol costs are negligible.
  - There is some form of randomness in the peer selection. Peers might be selected from the full set of nodes or from a smaller set of neighbors.
  - Due to the replication there is an implicit redundancy of the delivered information.

  Source: https://en.wikipedia.org/wiki/Gossip_protocol

SWIM

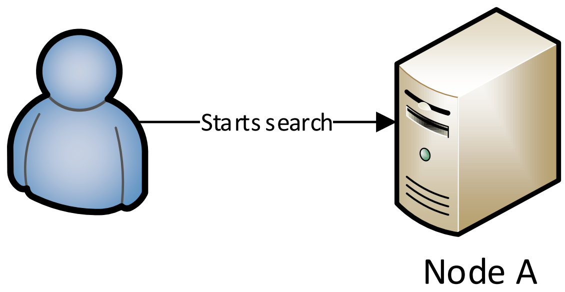
Gossip protocol – schema



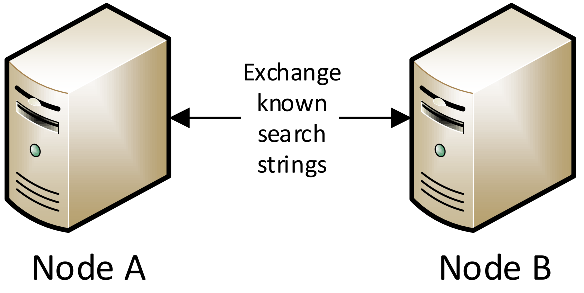
(c) Gossip-based approach, where peers operate in parallel, and each peer communicates with one or more randomly selected partner

https://jisajournal.springeropen.com/articles/10.1186/1869-0238-4-14

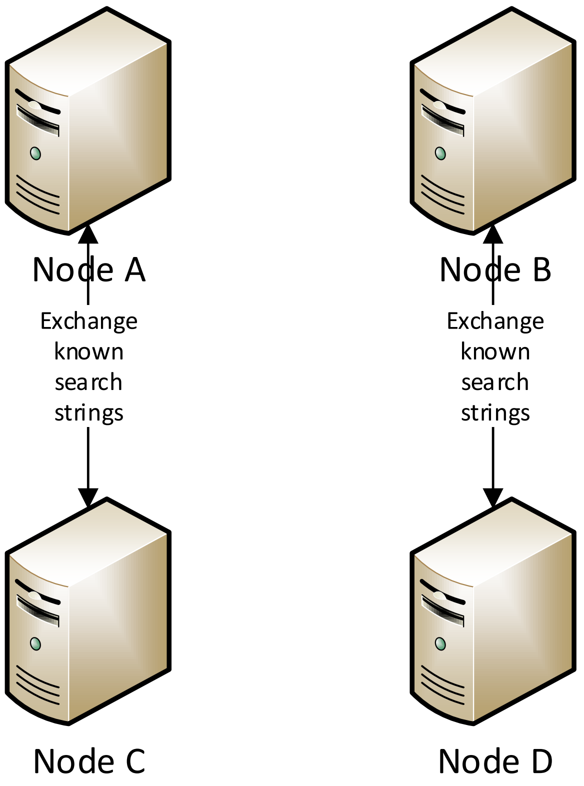
Gossip example – document search



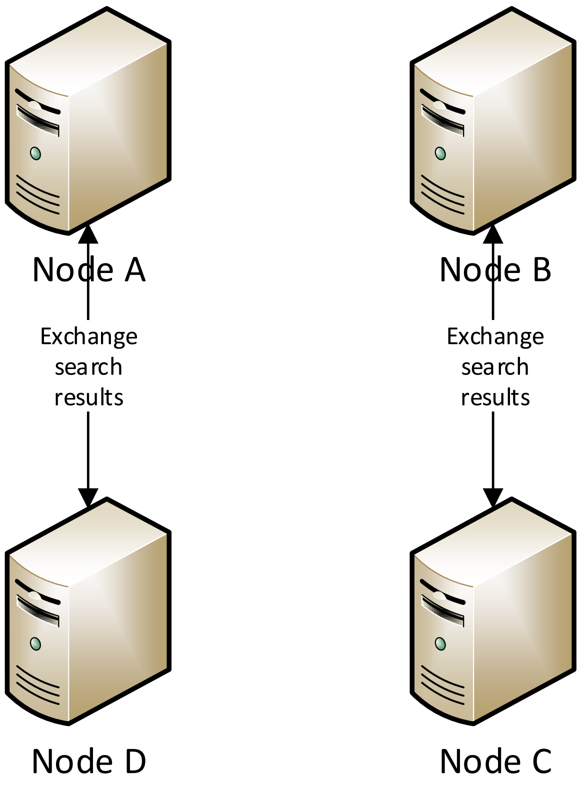
Gossip example – document search



Gossip example – document search

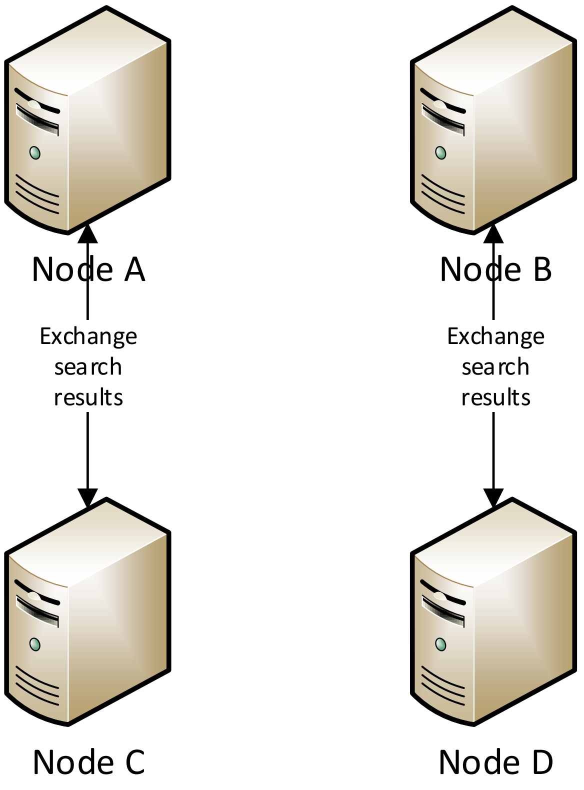


Gossip example – document search



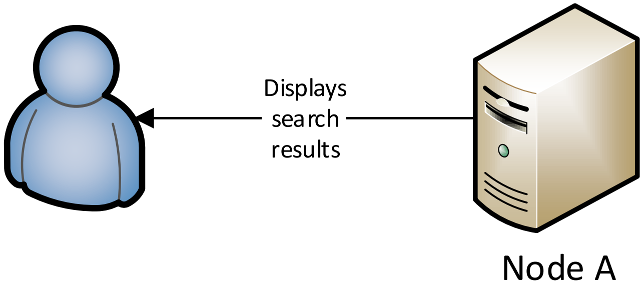
Hochschule Rosenheim
University of Applied Sciences

Gossip example – document search



Hochschule **Rosenheim**
University of Applied Sciences

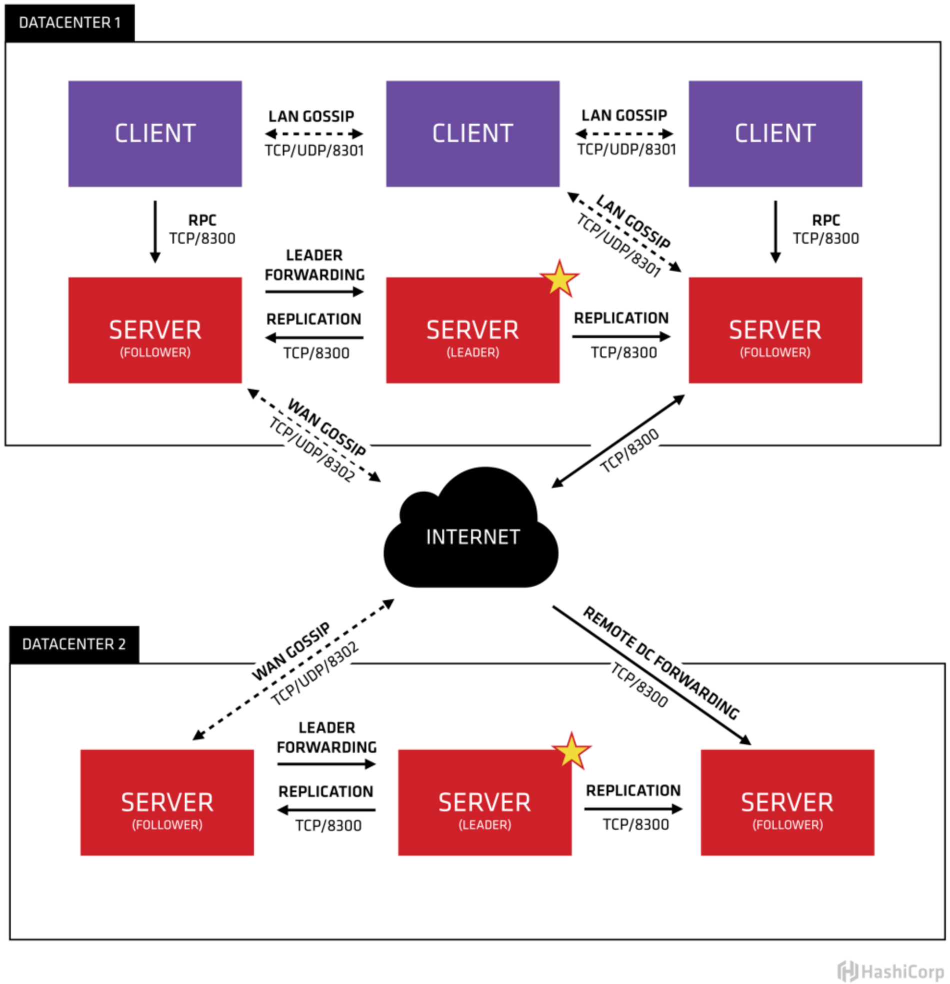
Gossip example – document search



Gossip example – document search remarks

- A search query should “age out” after a given time to reduce traffic
- If there are many search queries a maximum of data that may be exchanged during one “gossip” has to be defined
- Given a frequency of 10 gossips per second, a maximum of 30 rounds of gossip per search query and a network of 25.000 machines a query would take just about 3 seconds!

Consul architecture



Consul architecture – glossary

Term	Explanation
Agent	An agent is the long running daemon on every member of the Consul cluster. May be a client or server node (differences in WAN gossip)
Client	Forwards all RPC calls to a server. Only participates in LAN gossip.
Server	Expanded set of responsibilities compared to the client (Raft quorum, maintaining cluster state, …)
Datacenter	Defines a private, low latency, high bandwith areal

Consul architecture

- Expected to be 3 to 5 *server* instances per datacenter
- All nodes of a datacenter participate in the LAN gossip to determine which nodes are servers, doing heartbeats,…
- All nodes in the *server* mode are part of a single Raft peer set to elect a single leader per datacenter. The leader is responsible for all queries and transactions in the datacenter
- *Server* nodes are also participating in the WAN gossip pool. The WAN gossip pool exists to allow the datacenters to discover each other and is optimized for high-latency. Furthermore it's required for cross-datacenter queries.
- There's no data replication between data centers. Queries are always forwarded to the leader of the remote datacenter!

https://www.consul.io/docs/internals/architecture.html