Technische Hochschule Rosenheim
Technical University of Applied Sciences

Microservices

Service Discovery & API Gateways

Content

- Protocols
- Service Discovery
- API Gateways

Content - Protocols

- Gossip
- Raft

Content - Service Discovery

- Client-side service discovery
- Server-side service discovery
- DNS
- Key-value stores

Content - API Gateways

- Basics
- Load balancing strategies
- Failover techniques

Protocols

Protocols - Gossip

- Protocol to:
  - keep a cluster state in sync
  - manage the clusters health by constantly checking which nodes are available
- Used by Consul (Serf) based on Scalable Weakly-consistent Infection-style Process Group Membership Protocol (SWIM)

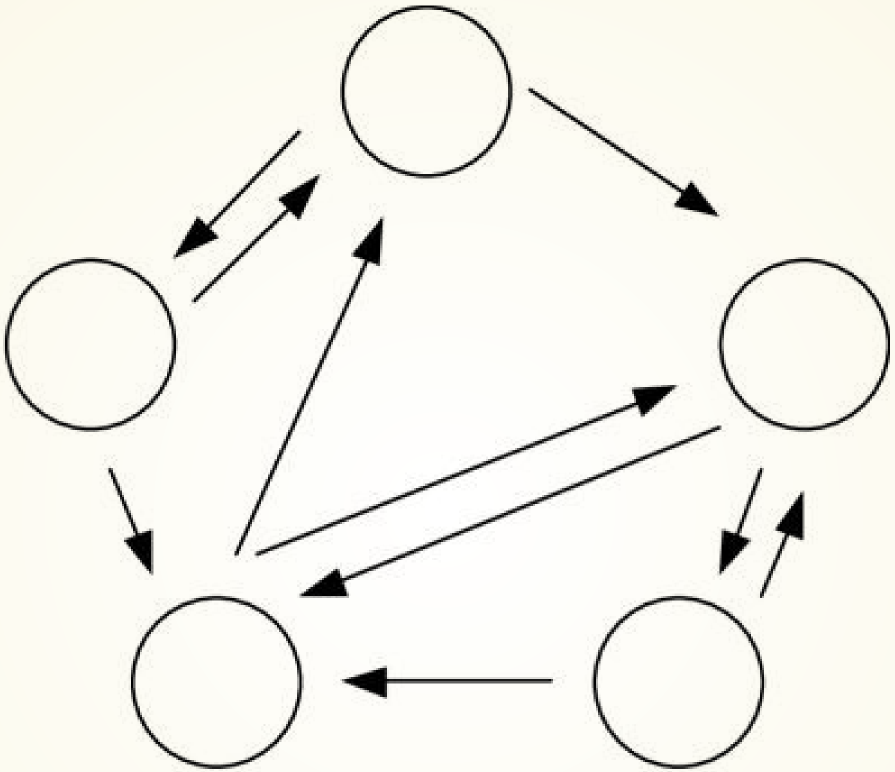
Gossip - principles - part 1

- 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

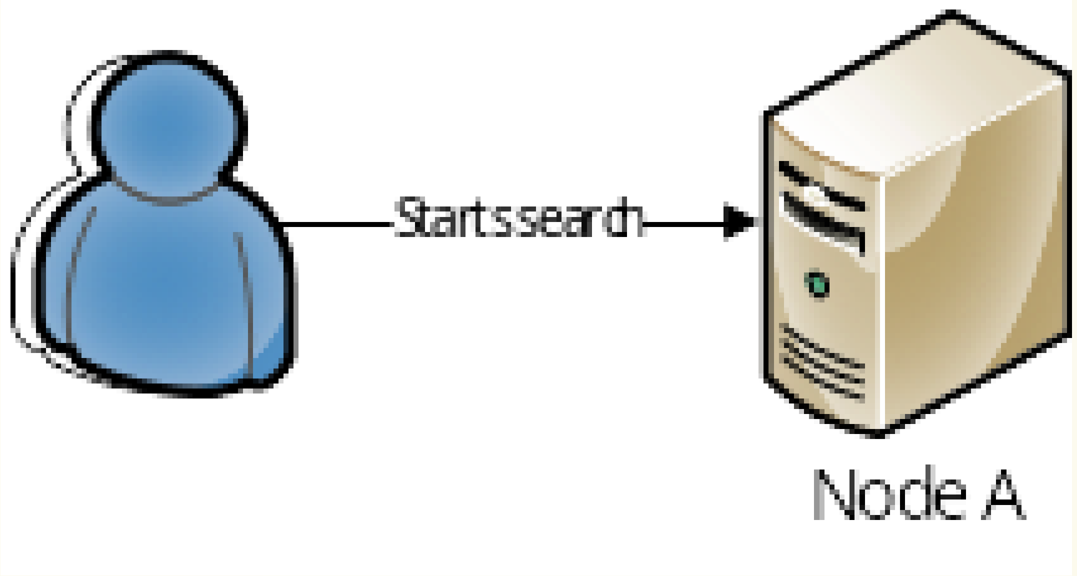
Gossip - principles - part 2

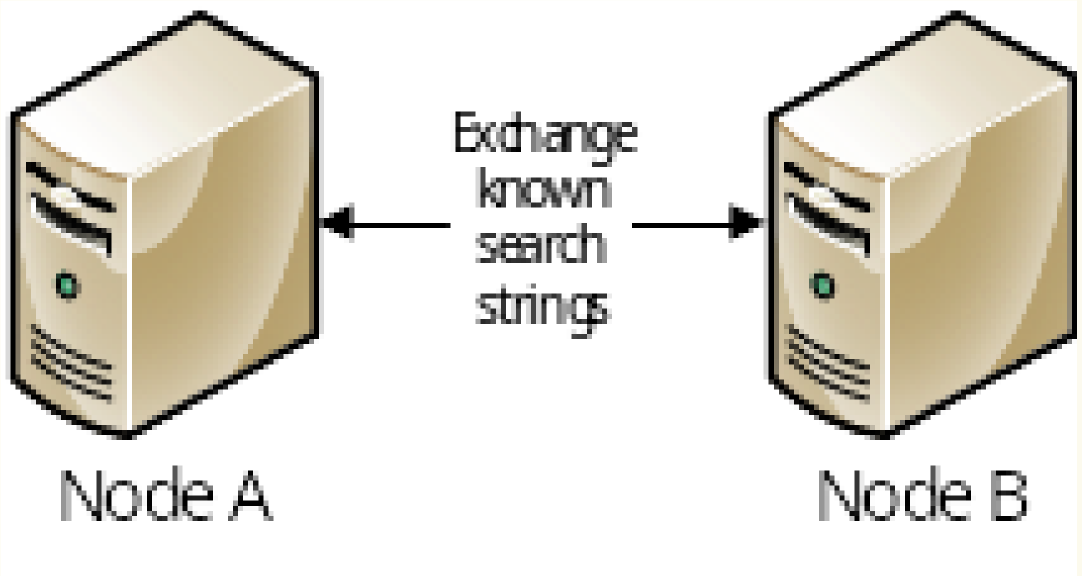
- The frequency of the interactions is low compared to typical message latencies so that the protocol costs are negligible.
- There is some form of randomness in the peer selection. Peers might be selected from the full set of nodes or from a smaller set of neighbors.
- Due to the replication there is an implicit redundancy of the delivered information.

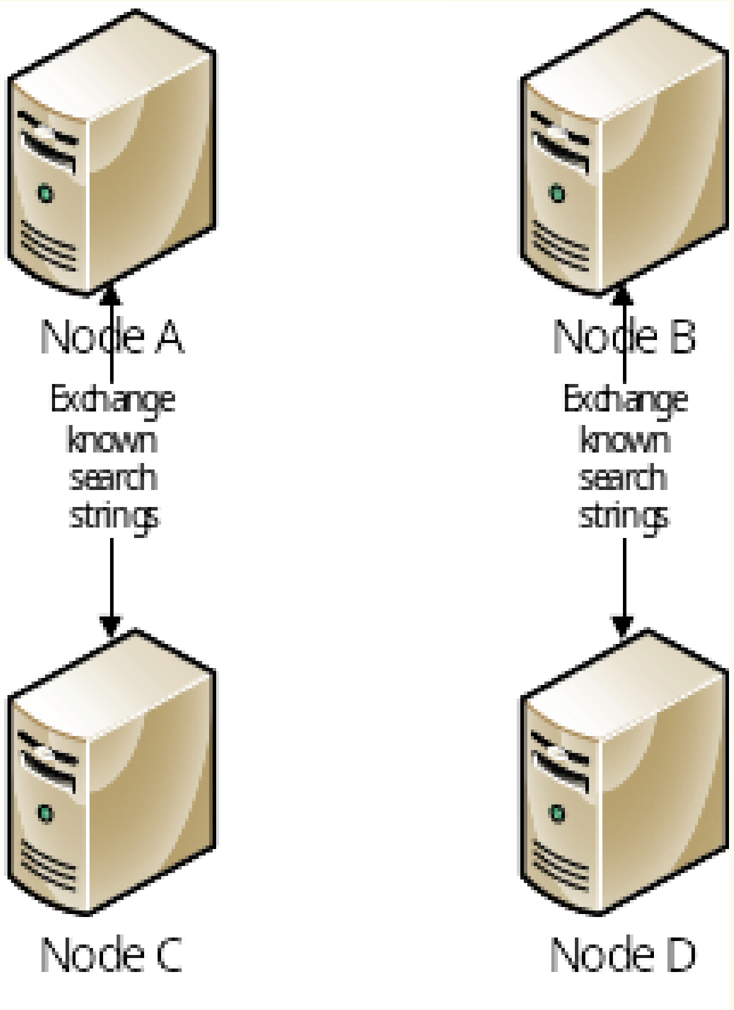
Source

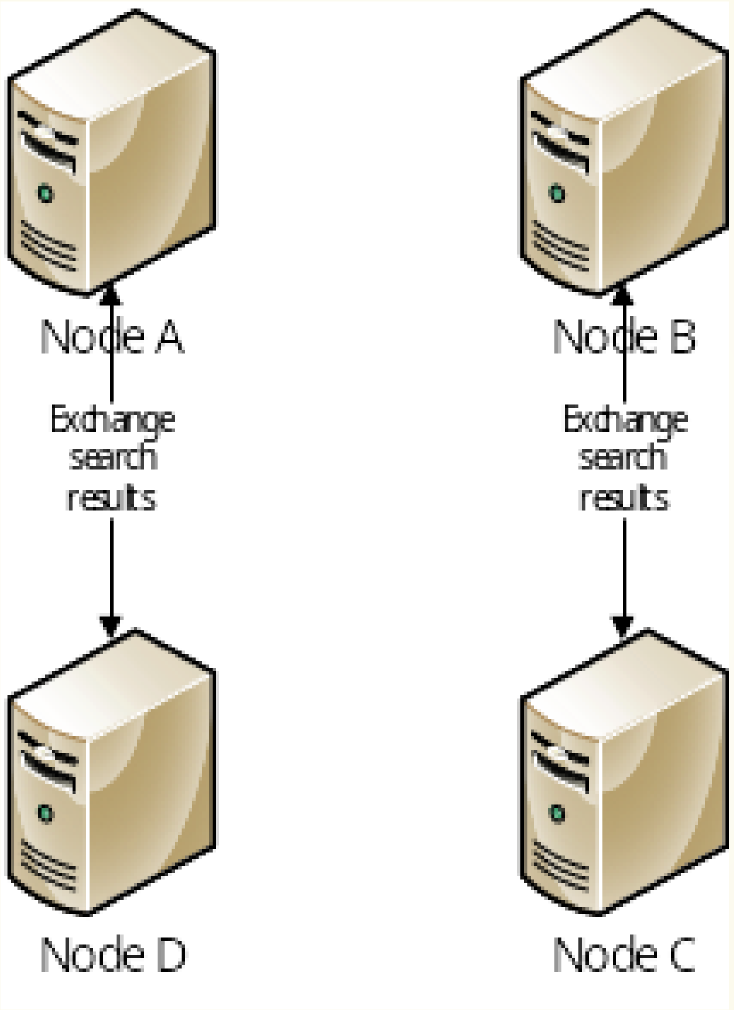


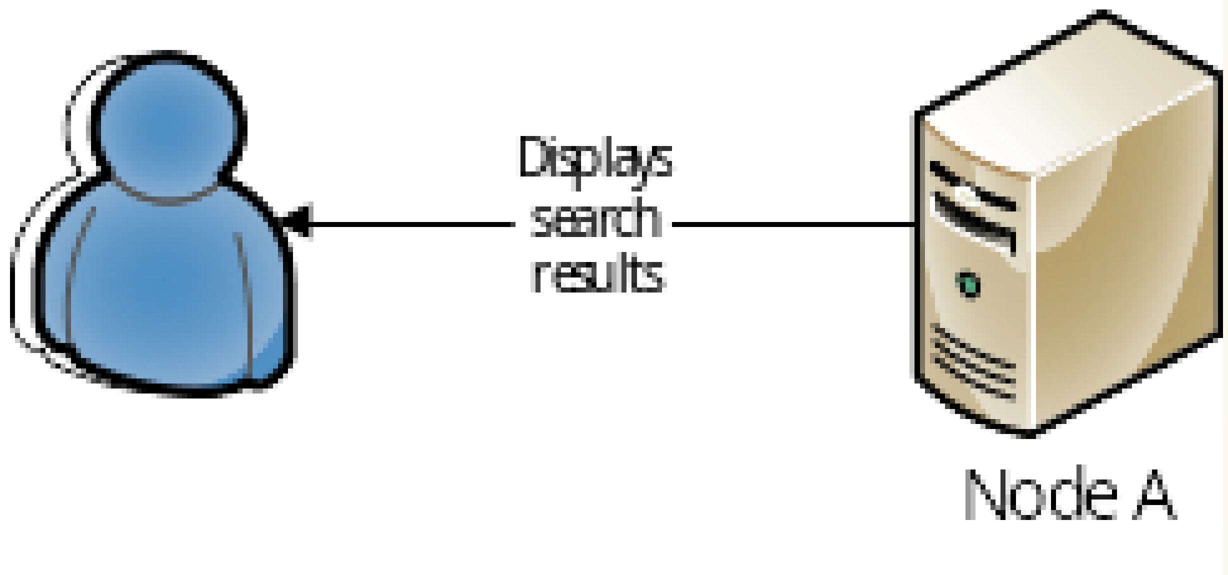
(c) Gossip-based approach,
where peers operate in parallel, and
each peer communicates with one or
more randomly selected partner


Starts search
Node A


Exchange known search strings
Node A
Node B


Node A
Exchange known search strings
Node B
Exchange known search strings
Node C
Node D


Node A
Node B
Exchange search results
Exchange search results
Node D
Node C


Displays search results
Node A

Gossip search sample - 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!

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Protocols - Raft consensus

Raft consensus basics

- Designed as an easier alternative to Paxos
- Uses leader election to achieve consensus
- Models a distributed state machine
  - Every node is a state machine
  - All nodes have to apply the same commands in the same order to stay in sync (same resulting state/transition)

Raft consensus basics

- Just one leader in a Raft cluster, all other nodes are followers
- Leader is responsible for the log replication to all followers
- Followers are expecting a heartbeat within a given timeout otherwise they suspect the leader failing
- If a leader fails a new leader is elected

Visualization https://raft.github.io/

Raft - leader election

- Leader election is started by a candidate server (a server that wasn’t contacted by the leader within the timeout period)
- Candidate increments the term number (serial for periods where a leader was elected) and proposes itself as the new leader and sends a message to all other servers requesting their vote

Raft - leader election

- If candidate gets a response with a term number at least as large as his current term number the election is defeated and the candidate is switching in follower mode
- If the candidate server gets a majority of votes he’s getting the new leader
- If neither happens (split vote) a new term is getting started (resulting in a new election)

Raft - log replication

- Leader replicates received requests (commands for the state machine) to all followers
- Leader appends the command to his log as a new entry and sends a `AppendEntry` to the followers
- When the leader receives confirmation of a majority of his followers he applies the entry to his state machine (request is considered committed)

Raft - log replication

- When a follower learns that an entry was applied by the leader he applies the entry to his local state machine
- In case of a leader crash the new leader enforces a replication of his log to all followers. To get a consistent state the leader compares his log with every log of the followers, takes the latest where they agree and replaces all following entries with his own

Raft - safety rules

- Election safety (at most one leader can be elected in a given term)
- Leader Append-Only (a leader can only append new entries to its logs - it can neither overwrite nor delete entries)
- Log Matching (if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index)

Raft - safety rules

- Leader Completeness (if a log entry is committed in a given term then it will be present in the logs of the leaders since this term)
- State Machine Safety (if a server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log)

Service Discovery basics

Service discovery mechanisms are required for multiple tasks in a microservice environment:

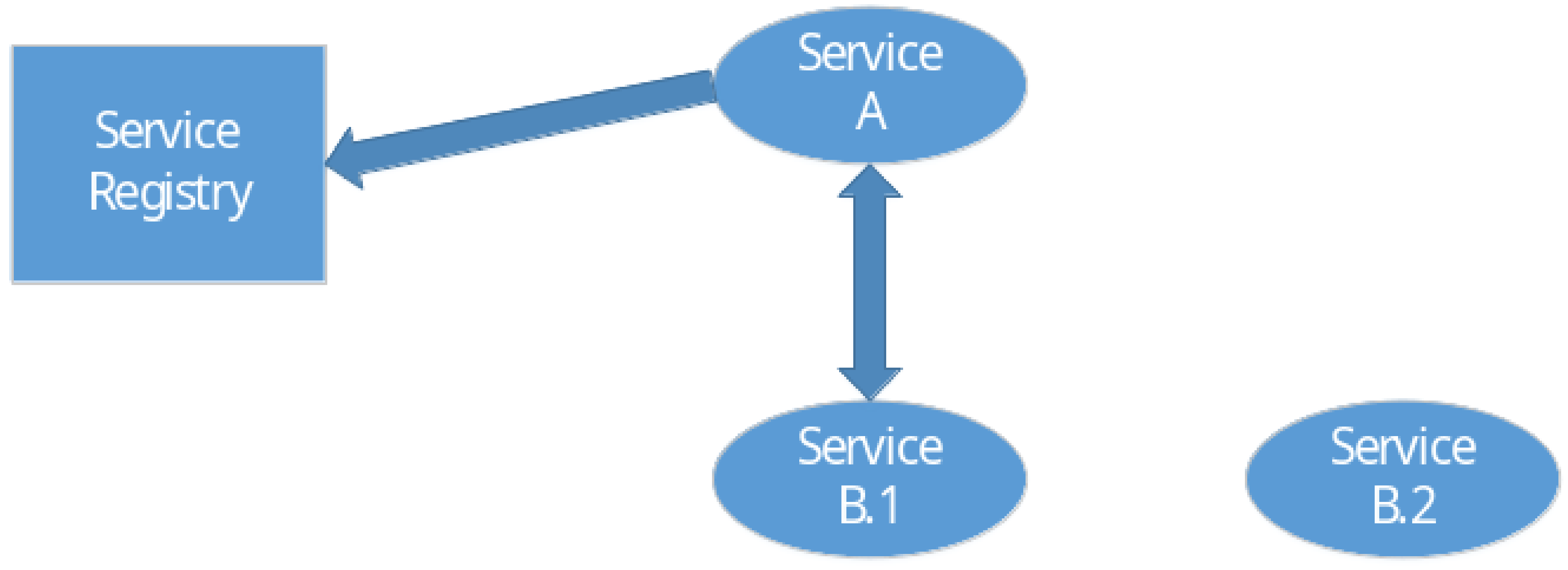
- server resolution for cross service communication
- dynamic load balancer configuration
- dynamic monitoring configuration
- …

Approaches

- Static configuration files (for the sake of completeness...)
- DNS based solutions (e.g. in Kubernetes with CoreDNS)
- Specialized products e.g. Eureka & Consul

Client-side vs. Server-side Service Discovery

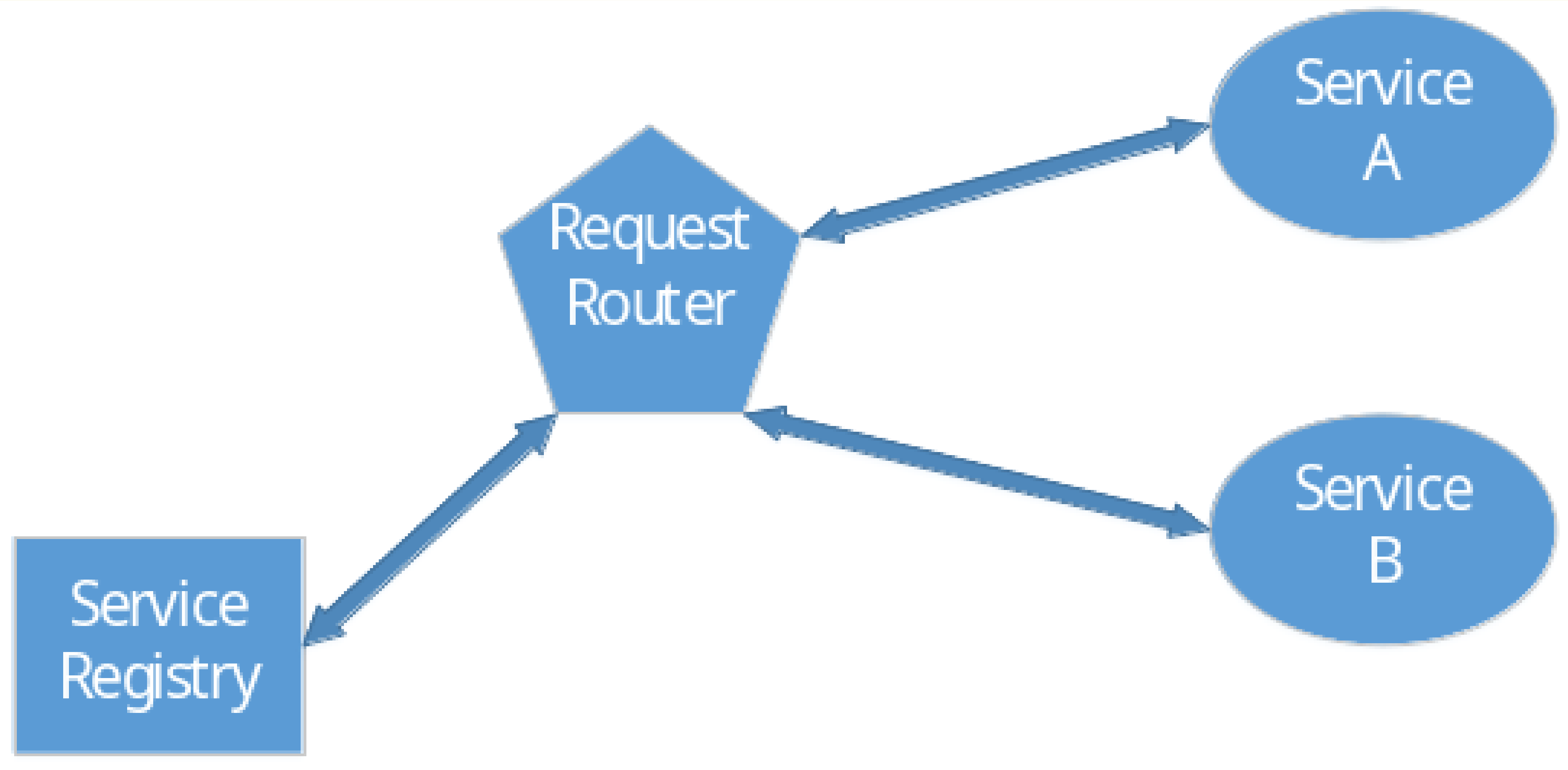
Client-side Service Discovery


Service Registry
Service A
Service B.1
Service B.2

Workflow

1. Service A/B.1/B.2 starts and registers itself at central service registry
2. Service A asks for one endpoint/all endpoints of Service B
3. Service Registry returns one/all endpoints to Service A
4. Depending on the implementation Service A decides which instance to call or calls the only it is aware of

Server-side Service Discovery


Service A
Request Router
Service B
Service Registry

Workflow

1. Service A/B starts and registers itself at central service registry
2. Service A sends requrest to central router (e.g. an API Gateway)
3. Gateway is redirecting call to concrete endpoint like a proxy
4. Gateway returns the response it got from the concrete endpoint to the original caller

Service registration

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

Service registration - 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, Nomad

DNS based Service Discovery

DNS - viable record types - part 1

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)

DNS - viable record types - part 2s

Record name	Explanation
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)

Source

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 it's also possible to store the port of the service in a SRV record
- Every instance has to register itself at a DNS server or a 3 rd 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 sample	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 e nvironment

DNS as service registry - considerations

- Relatively easy to implement
- No special software/libraries required
- TTL of entries might lead to stale entries
- DNS caching
- Requires special DNS server implementation to support dynamic registration

Key-value stores for Service Discovery

Classical ones:

- ZooKeeper
- etcd

Specific for Microservices:

- Consul
- Eureka

Classical key-value stores

- Developed as distributed configuration stores
- Hierarchical structured
- Normally offer some kind of "watches" or "subscriptions"

Microservice specific ones

- Implement specific domain knowledge (special entities for services and endpoints)
- Offer possibilities to register e.g. health checks to ensure that registered services are available
- Some are also offering configuration stores (e.g. Consul)
- Various APIs (e.g. HTTP or DNS)

Load balancing basics

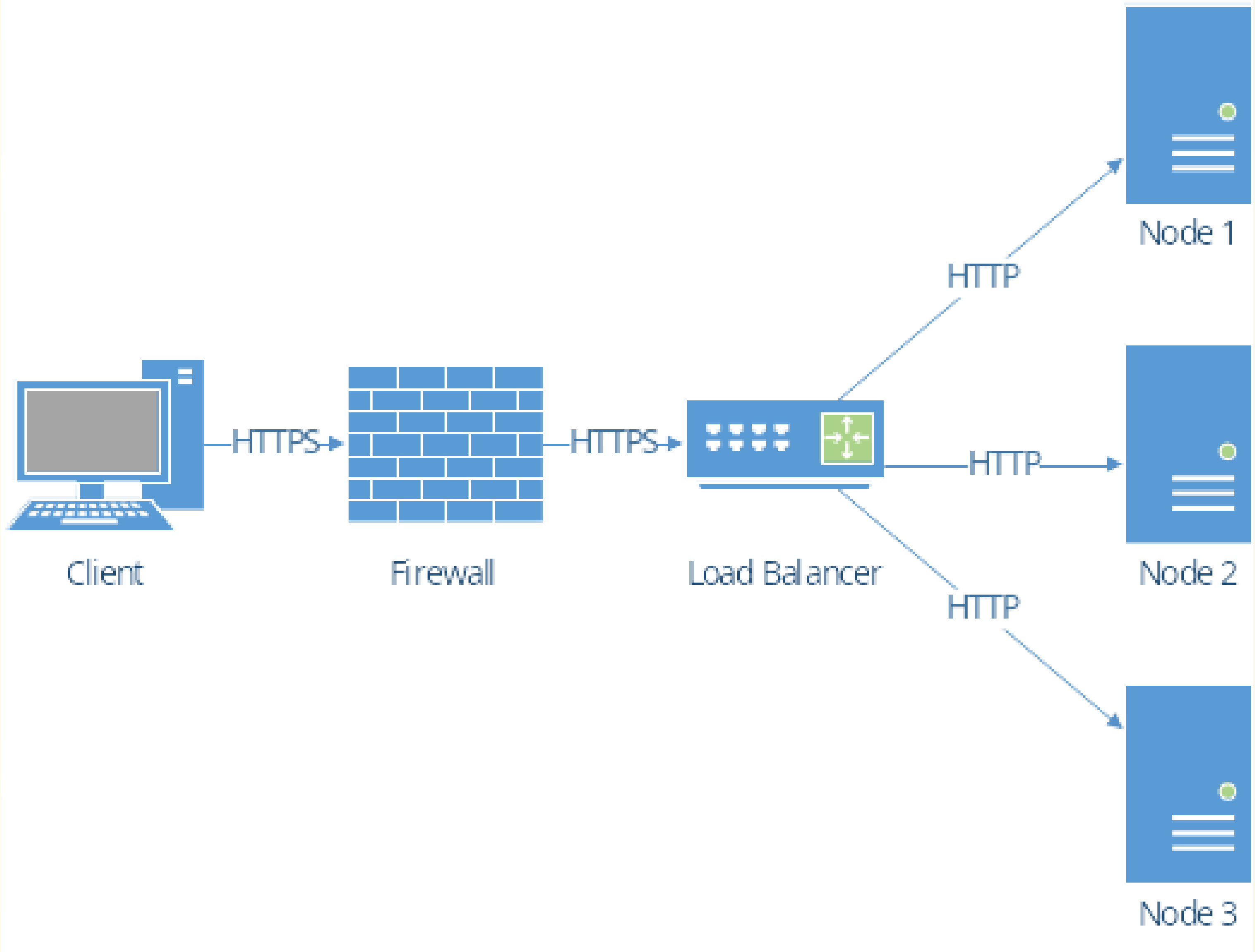
- Load Balancers are needed to avoid single point of failures
- Load Balancers distribute calls sent to them to one or more instances
- Load Balancers keep track of their known backends to avoid errors when a services is no longer healthy

Load balancing basics

- Optionally they have additional features like SSL termination
  - Only one or a few servers where certificates have to be exchanged when a new certificate is created
  - No special handling required in the services behind the load balancer
  - Admins have to take care that communication between load balancer(s) and nodes is safe (e.g. VLANs)

Load balancing basics

- In Microservice environments it’s essential that the load balancer(s) can be reconfigured dynamically (e.g. in combination with etcd or Consul)
- When you’re able to scale your microservice instances but not the persistence layer of them you’re just moving the single point of failure one layer backwards!


HTTPS
HTTPS
HTTP
HTTP
HTTP
Client
Firewall
Load Balancer
Node 1
Node 2
Node 3

Load balancing strategies

- Round-robin
- Weighted round-robin
- Least connection
- Weighted least connection
- Agent Based Adaptive Load Balancing
- Chained Failover (Fixed Weighted)
- Weighted Response Time
- Source IP Hash
- ...

(Weighted) Round-robin

- There are many Round-robin algorithms
- The simplest one is to use a FIFO queue to keep track of all available backends
  1. Dequeue
  2. Relay request
  3. Enqueue

(Weighted) Round-robin

- Results in max-min-fairness (the longest waiting requests gets the highest priority)
- The weighted round robin algorithm gives every backend a weight and the scheduler takes the weights into account to prefer servers with a higher weight before servers with a lower weight (e.g. used for quality of service (QoS))

(Weighted) Least connection

- In contrast to round-robin the least connection algorithm takes the load of every node in account.
- The least connection algorithm relays an incoming request always to the node which has the lowest count of active connections.

(Weighted) Least connection

- This way nodes with a higher performance handle more requests than nodes with lower performance without the need to configure weights.
- The weighted least connection variant enables the administrator the give nodes a weight. These weights are considered when two nodes serve the same count of active connections and the node with the higher weight is considered first.

Agent Based Adaptive Load Balancing

- Every node has an local agent installed which reports real time data to the load balancer (e.g. CPU usage, memory allocation,...)
- Load balancers takes load of every node into to account when a new request has to be relayed
- Can be combined with weighted round-robin or weighted least connection algorithms
- Used e.g. in Windows Terminal Server (RDS role after Server 2003)

Chained Failover

- All backend nodes are in a predefined chain
- Whenever the first node can’t handle/accept another request the next node in the chain is taken into account and so on
- Not a real load balancing protocol!

Weighted Response Time

- Kind of health check done by the load balancer(s)
- Uses the response time of the health check to determine the fastest server currently available
- Whenever a node is under heavy load the response times will be longer than the response times of a node with least load.
- Avoid overload of nodes.

Source IP Hash

- Algorithm creates a hash of source and destination IP (unique hash key)
- Hash key is used to determine to which node the request should be forwarded

Source IP Hash

- When the same client sends another request the hash key can be regenerated and the client gets forwarded to the same node
- Useful for stateful services (don’t do that in microservices!) when nodes aren’t able to sync session information because a client always gets relayed to the same node (as long as its source IP does not change)

Network failover

- But wait! If I have 1 load balancer what happens if this load balancer fails?
- Possible solution: multiple load balancers with multiple DNS A/AAAA records to balance the load of the load balancers -> but DNS does not check for availability and there are the caches...

Network failover

- Better solution: configure network based failover:
  - Common address redundancy protocol (CARP)
  - Gateway Load Balancing Protocol (GLBP) (just for routers)

Common address redundancy protocol (CARP)

- Enables multiple hosts in the same LAN to share a set of IP addresses
- Available on BSD and Linux based hosts
- Master-slave (or more polite active-passive) based
- One master per group of redundancy
- Each group of redundancy shares one virtual IP
- A server maybe member of multiple groups of redundancy

Common address redundancy protocol (CARP)

- Every server needs a second IP address for communication (best practice is to configure two IP addresses: one for LAN communication and one for the communication between all members of the group of redundancy e.g. heartbeats)
- Whenever the master fails a slave takes over and answers all incoming requests
- Can be combined with DNS round robin e.g. two groups with two members each to ensure that always two load balancers are available

Gateway Load Balancing Protocol (GLBP)

- Proprietary protocol created by Cisco for redundant routers
- Allows weighting parameter to be set
- Based on the weights (in a virtual router group) ARP requests will be answered with MAC addresses pointing to different routes
- Balances in round-robin fashion by default

Gateway Load Balancing Protocol (GLBP)

- Elects one Active Virtual Gateway (AVG) for each group
- AVG assigns every listener (and itself) virtual MAC addresses which enables Active Virtual Forwarders (AVF)
- Each AVF is responsible to forward packages sent to its virtual MAC address

API Gateways

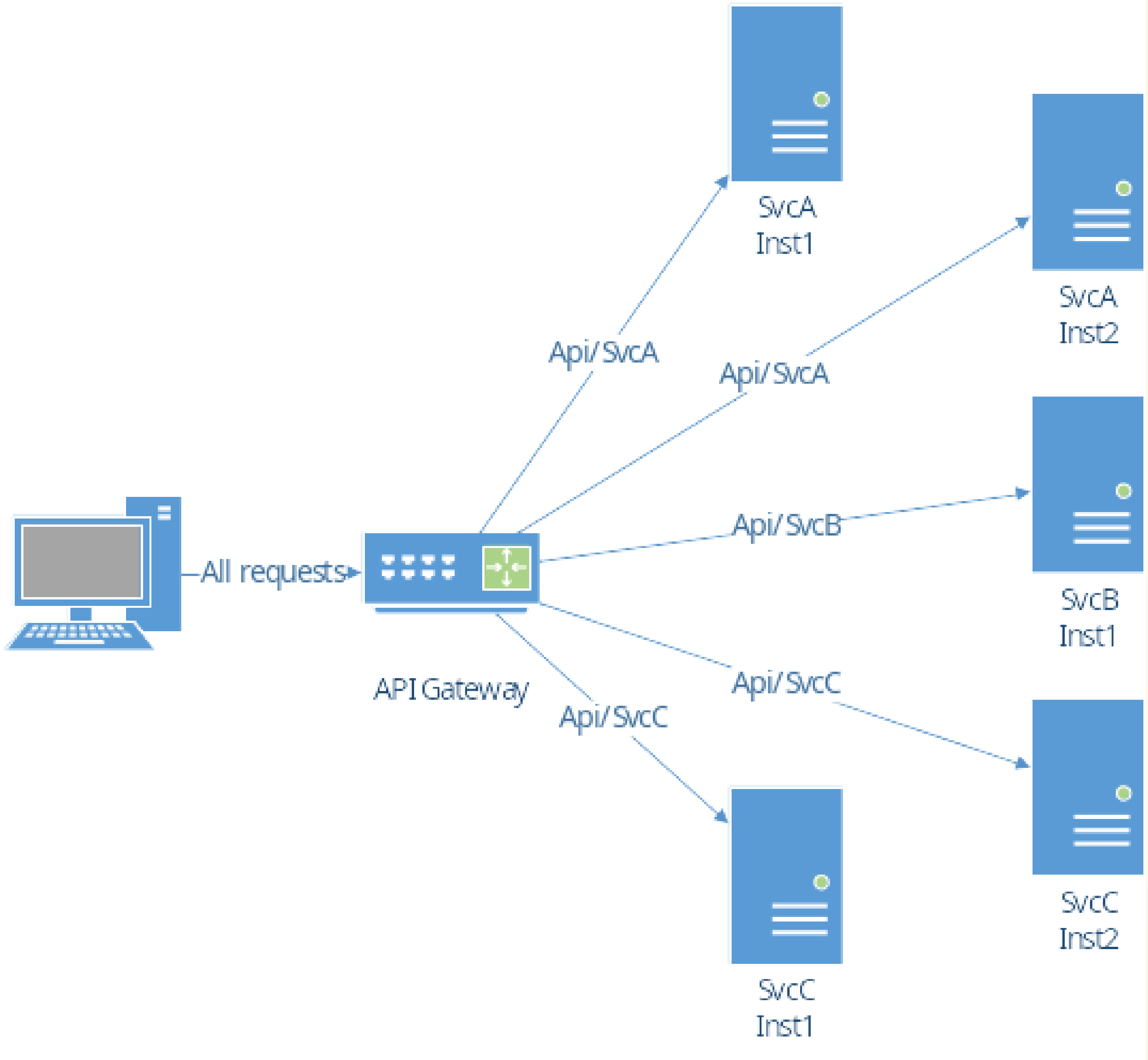
- API Gateways are a crucial part of every microservices environment
- API Gateways enable a microservices environment to scale by implement load balancing (e.g. round-robin based)
- Access to specific services is managed by:
  - DNS-Host per Service (A/AAAA/CName e.g. ServiceA.my-domain.com)
  - Virtual Routes (e.g. gateway.my-domain.com/ServiceA)

API Gateways

- API Gateways are the single entry point for your whole application
- API Gateways are a kind of server-side service discovery (usually just for client apps but it’s possible to use it also for cross service calls)

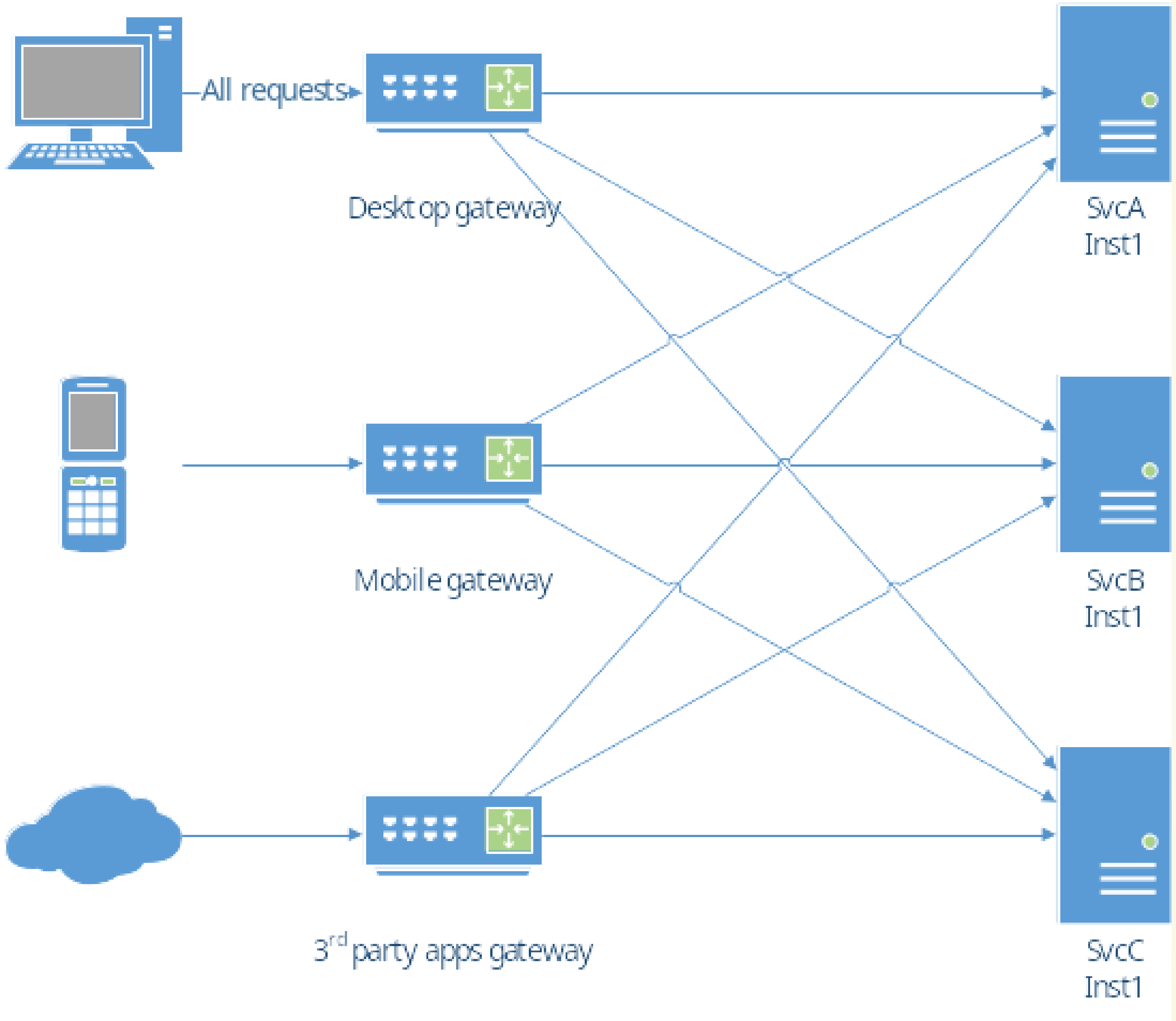
API Gateways

- They also hide implementation details by optionally aggregating all internal APIs to one (or more in case of Backends for frontends) in the point of view of the client app(s)
- In the case of custom API Gateways the gateway may also execute calls to multiple services and aggregate the responses answering to the client request


All requests
API Gateway
Api/SvcA
Api/SvcA
Api/SvcB
Api/SvcC
Api/SvcC
SvcA Inst1
SvcA Inst2
SvcB Inst1
SvcC Inst2
SvcC Inst1

Backend for frontends

- Configure a API Gateway per kind of frontend e.g.
  - One for your web app
  - One for your mobile app
  - One for all 3 rd party applications (public API)
- The backends for frontends-pattern ensures the optimal API for every kind of application (e.g. gRPC gateway for desktop apps but RESTful API for web apps)


All requests
Desktop gateway
Mobile gateway
3rd party apps gateway
SvcA Inst1
SvcB Inst1
SvcC Inst1