FOCUS: Scalable Search Over Highly Dynamic Geo-distributed State

Azzam Alsudais
Mohammad Hashemi
Eric Keller

Zhe Huang, Bharath Balasubramanian
Shankaranarayanan Puzhavakath Narayanan
Kaustubh Joshi

IEEE ICDCS 2019 – Dallas, TX, USA
July 9, 2019

Why do systems need to find nodes?

Use Cases

Cloud Management

Use Cases

Cloud Management

VM Provisioning

Use Cases

Cloud Management

VM Provisioning

VM Migration

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

Geo-distributed VNF

Service Chain Placement

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

Geo-distributed VNF

Service Chain Placement

Required information is assumed available

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

Geo-distributed VNF

Service Chain Placement

Required information is assumed available

But **HOW** is node information collected?

Outline

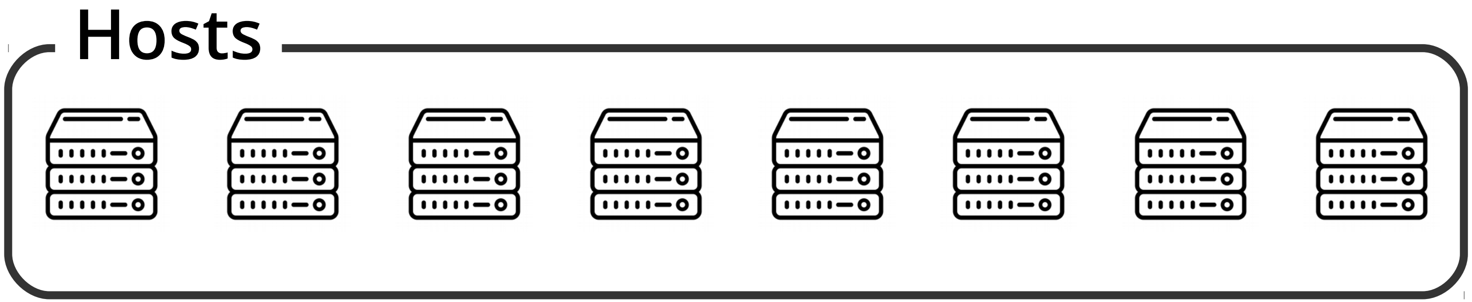
- How do systems find nodes?
- Limitations of current approaches
- FOCUS design
- Evaluation
- Conclusion

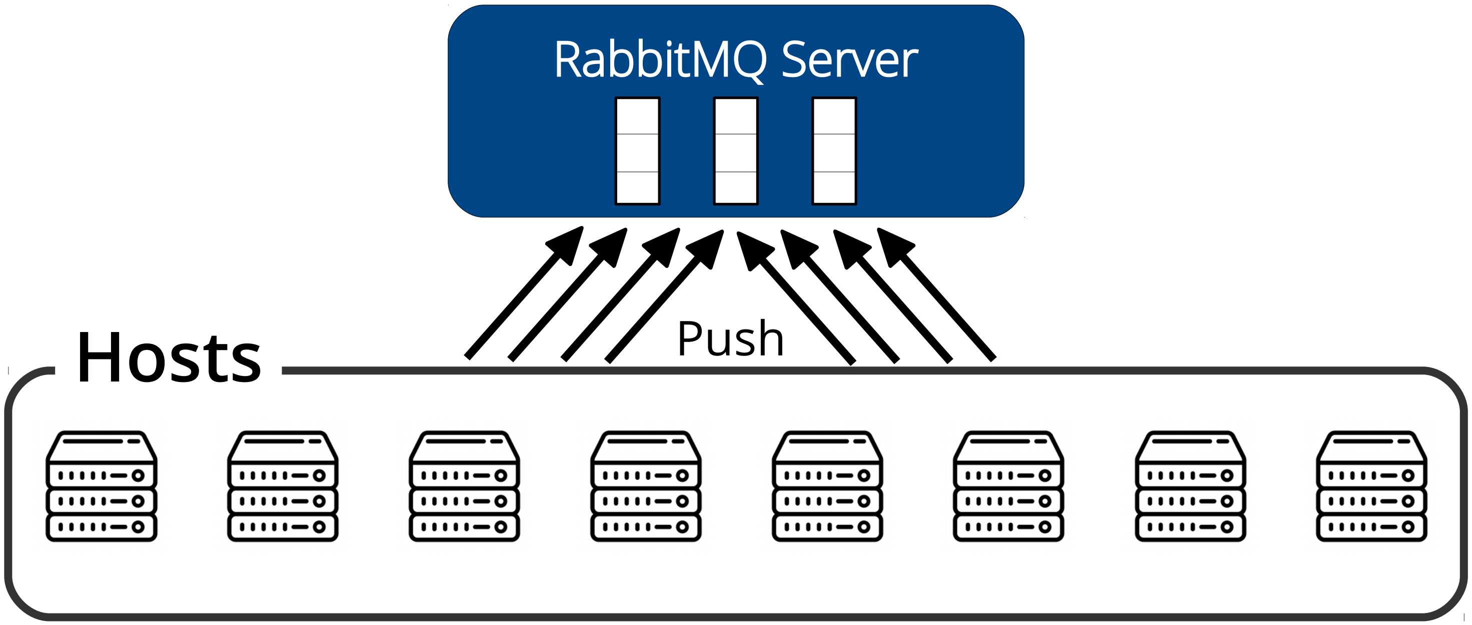
Looking Under The Hood

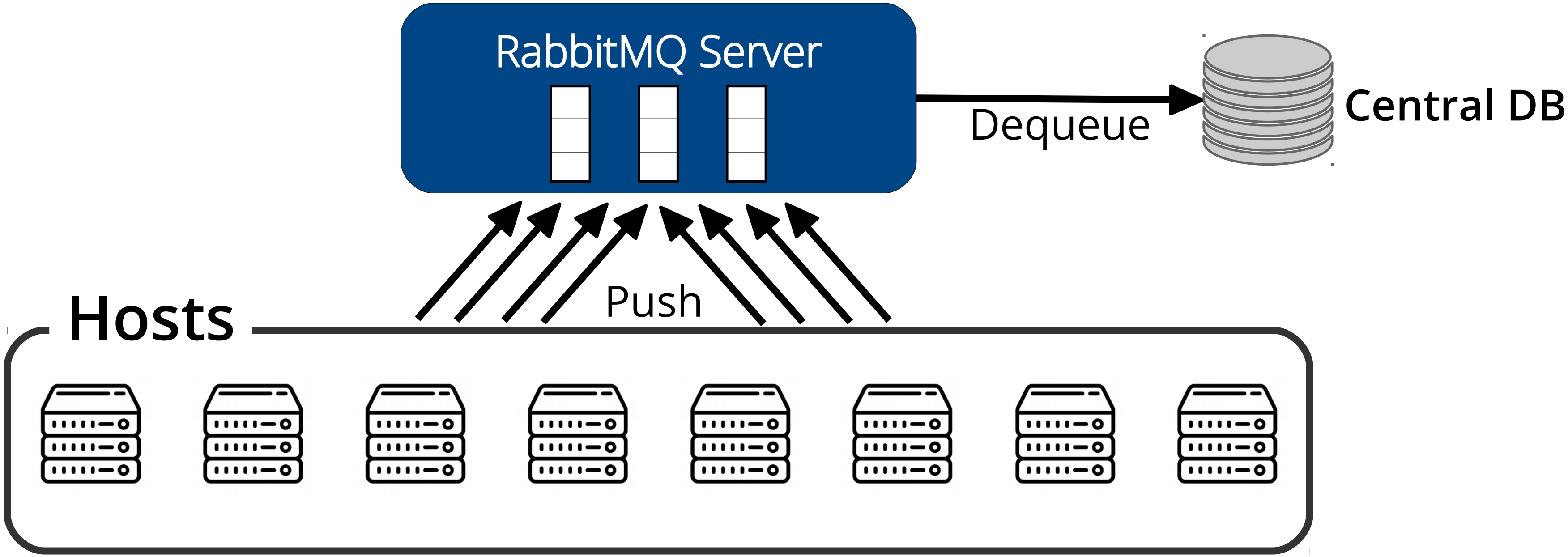
How do systems search for nodes?

Node finding in openstack™

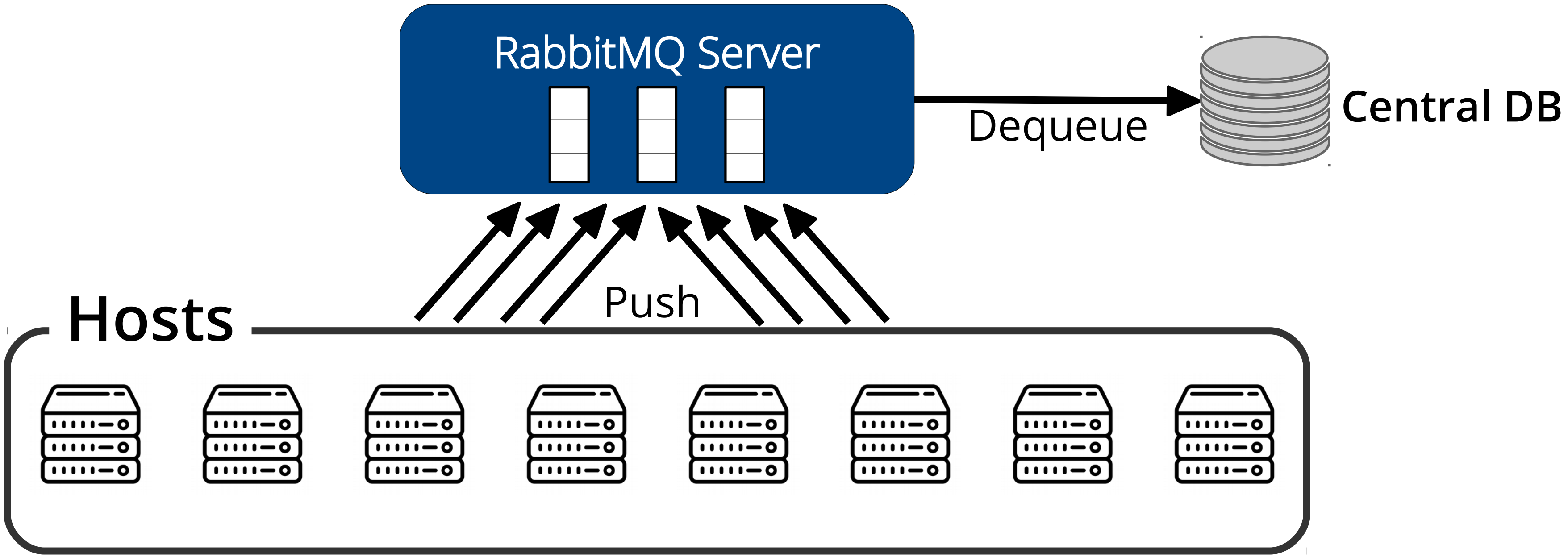
openstack™

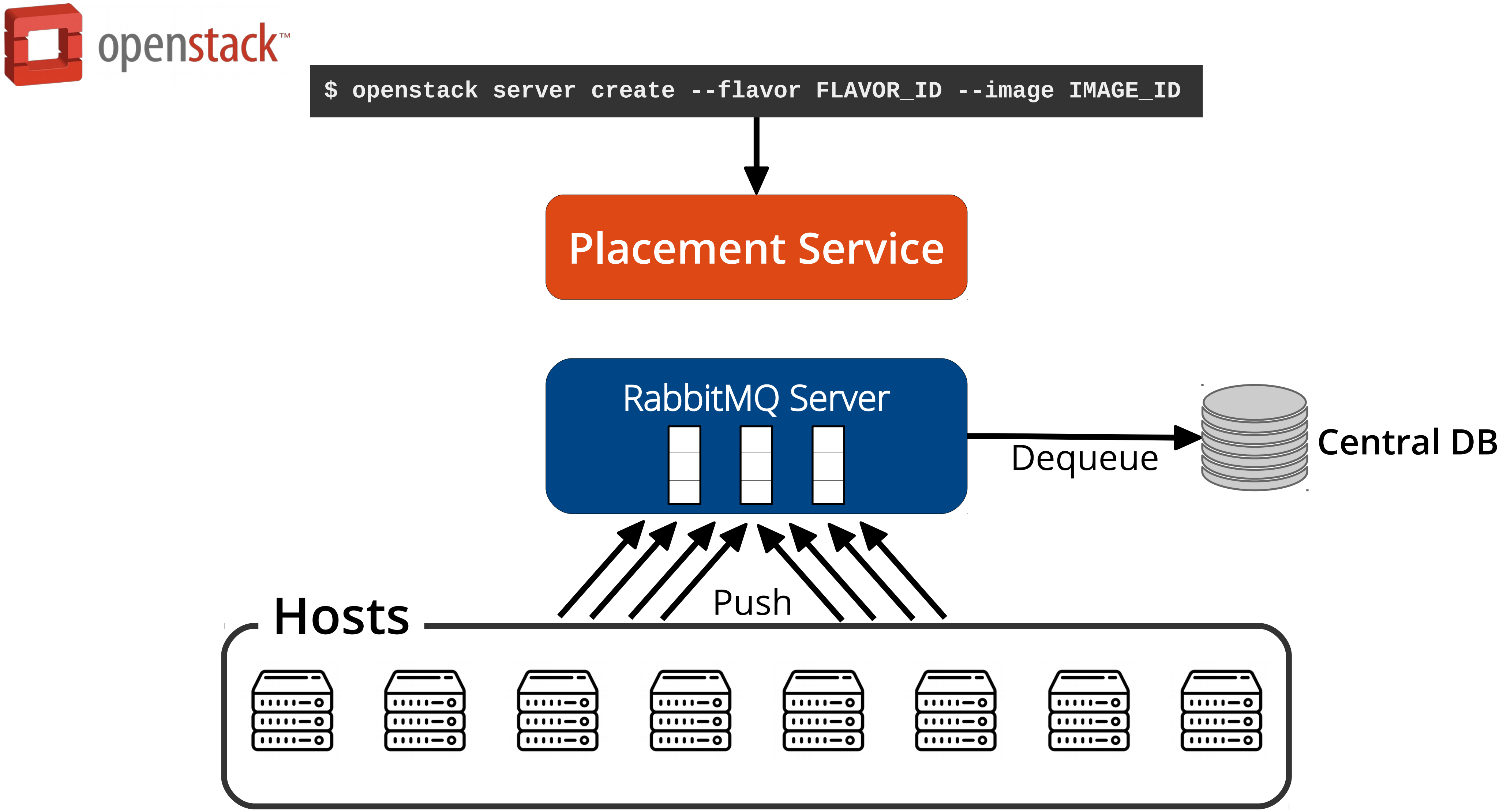


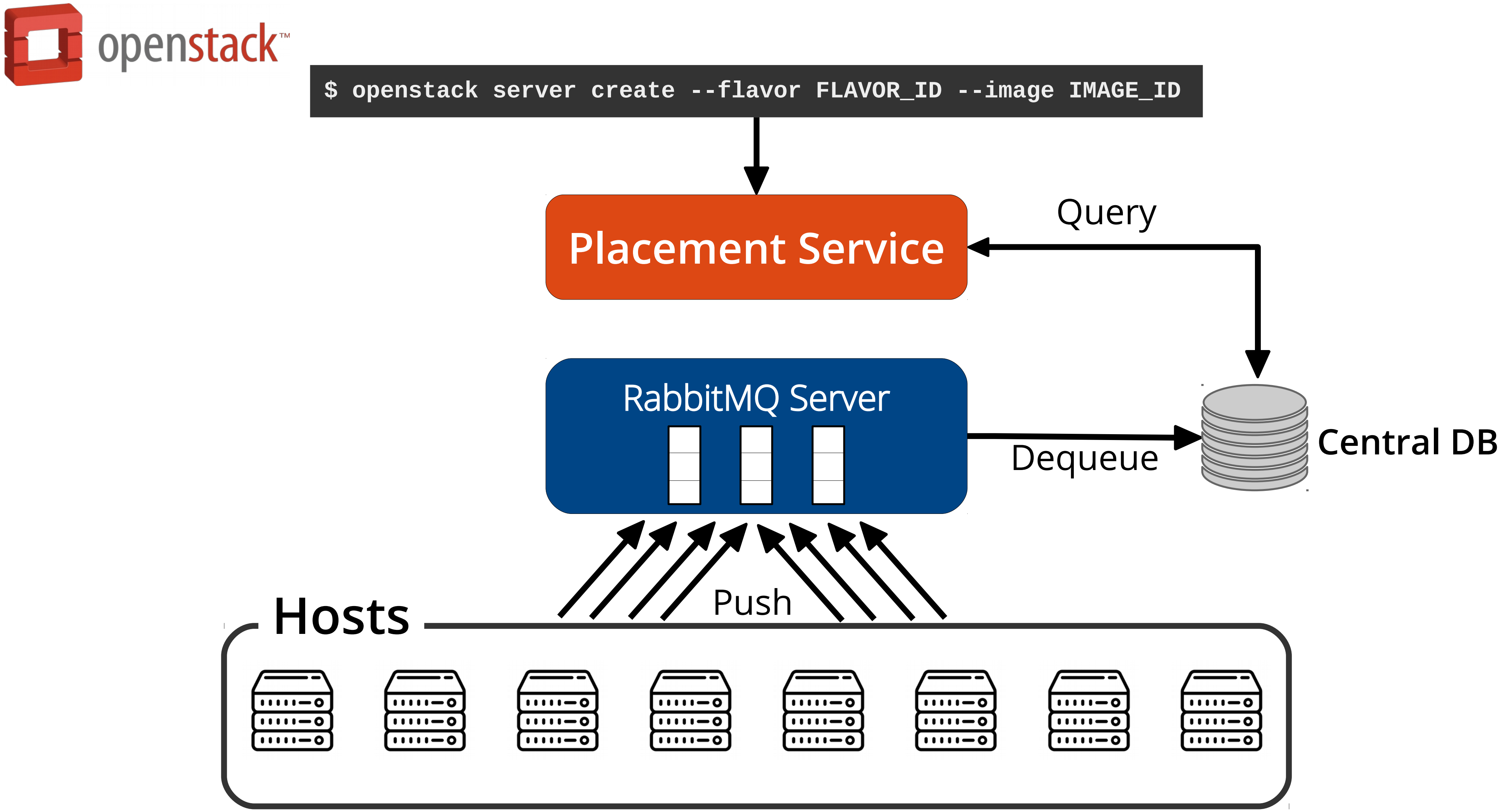

RabbitMQ Server
Push
Hosts

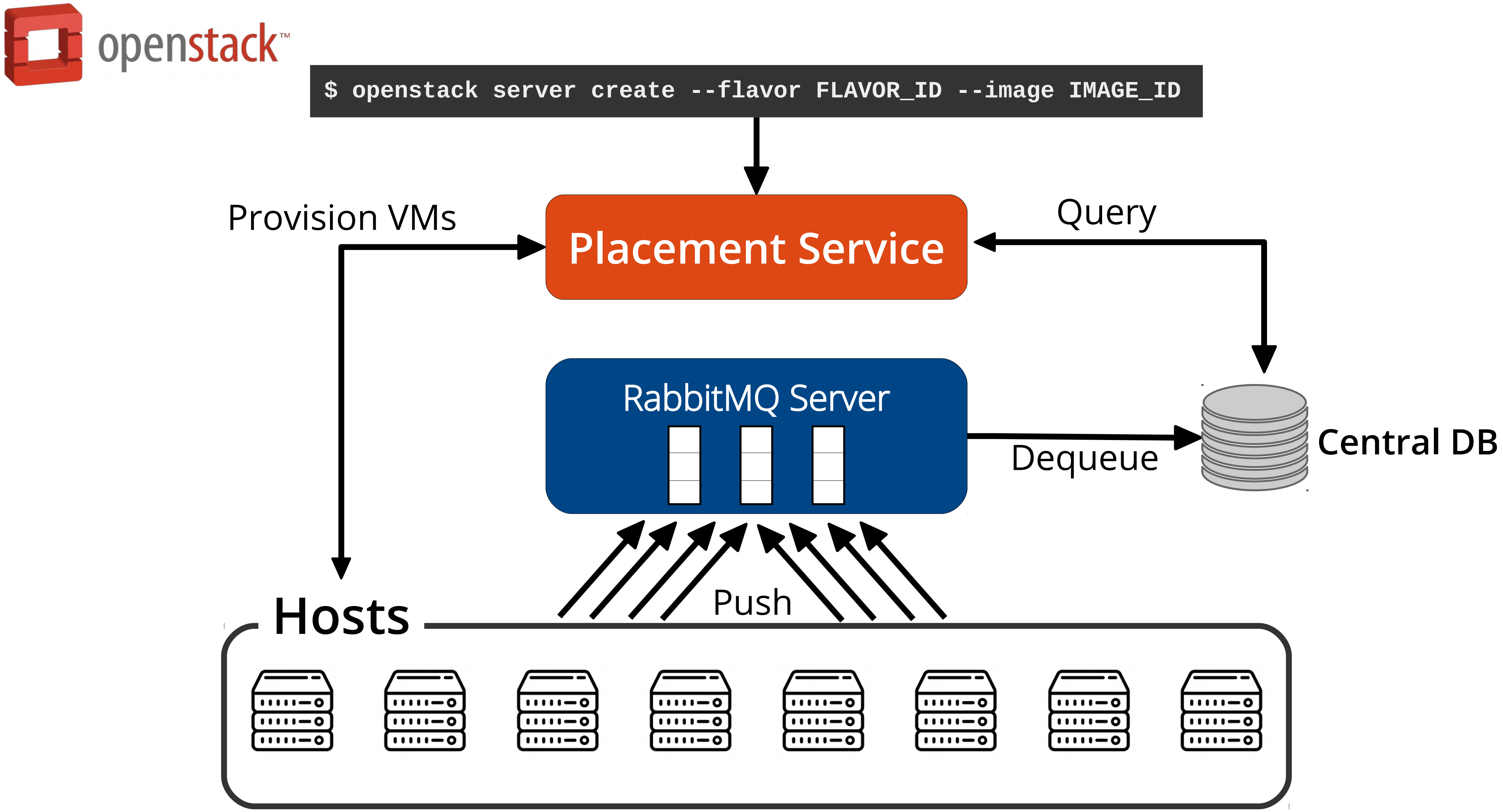

RabbitMQ Server
Dequeue
Central DB
Push
Hosts

```
$ openstack server create --flavor FLAVOR_ID --image IMAGE_ID
```

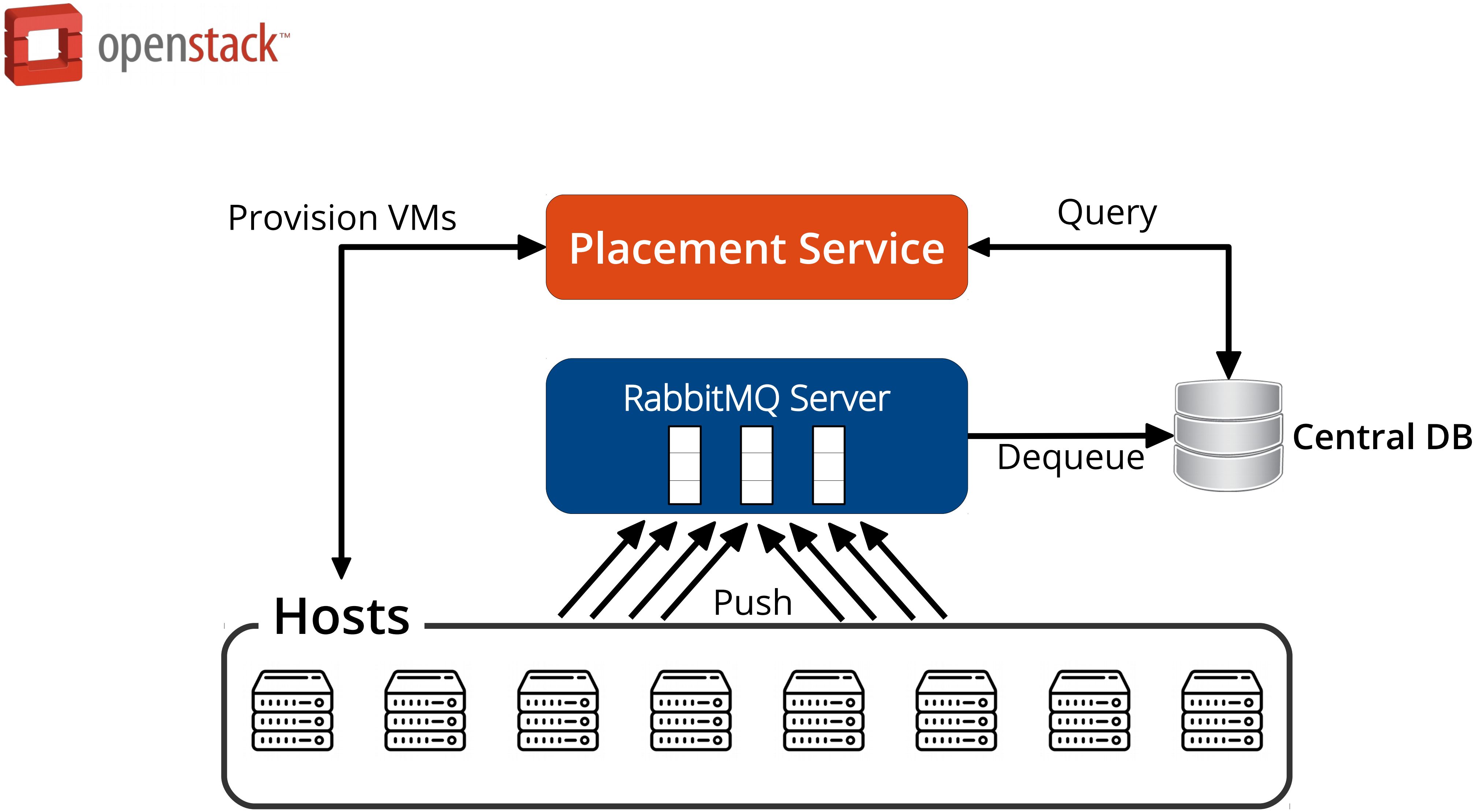



openstack
$ openstack server create --flavor FLAVOR_ID --image IMAGE_ID
Placement Service
RabbitMQ Server
Dequeue
Central DB
Push
Hosts

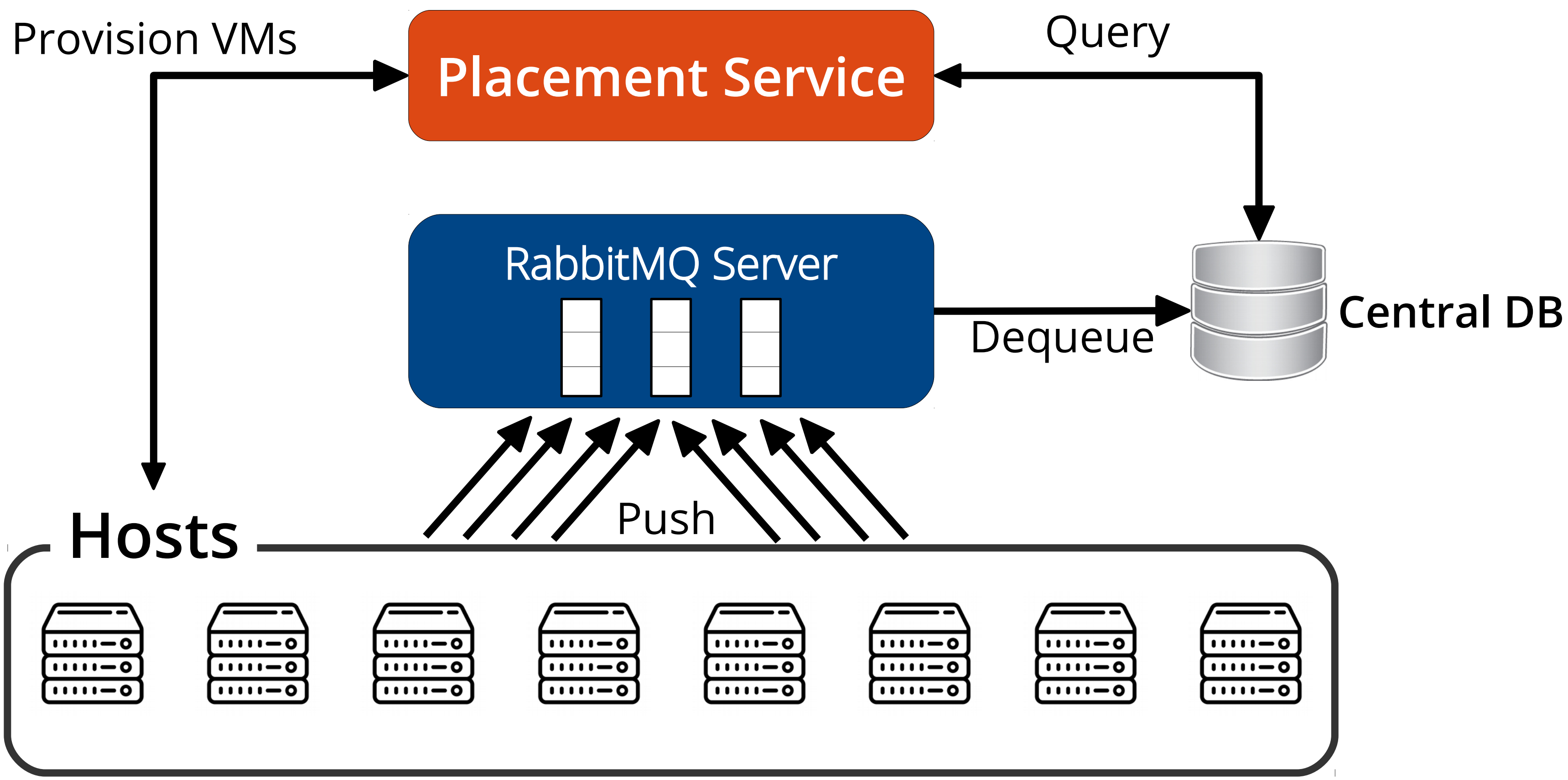

openstack
$ openstack server create --flavor FLAVOR_ID --image IMAGE_ID
Placement Service
Query
RabbitMQ Server
Dequeue
Central DB
Push
Hosts


openstack
$ openstack server create --flavor FLAVOR_ID --image IMAGE_ID
Provision VMs
Placement Service
Query
RabbitMQ Server
Dequeue
Central DB
Push
Hosts

Limitations of Current Approaches

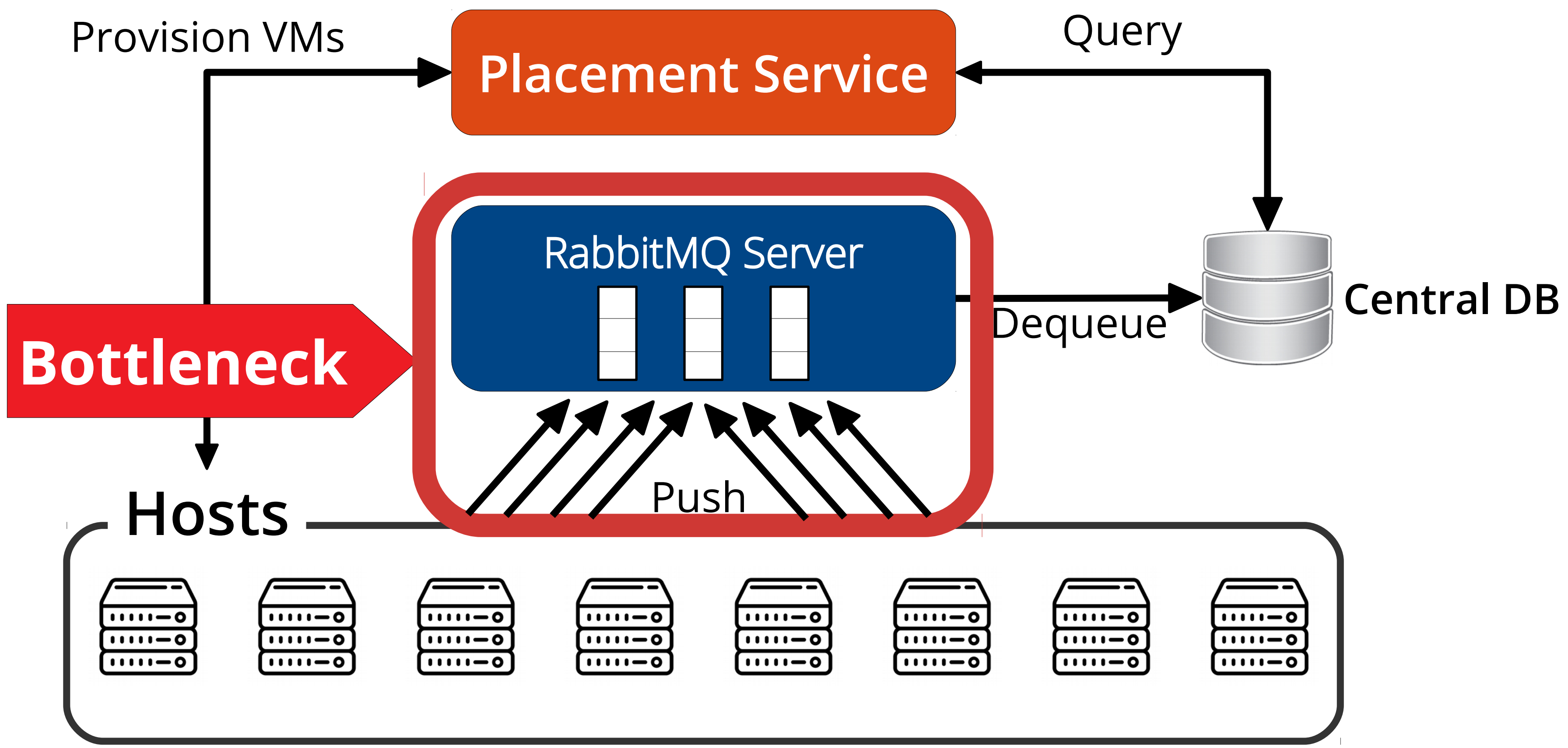

openstack
Provision VMs
Placement Service
Query
RabbitMQ Server
Dequeue
Central DB
Hosts
Push

Hard to scale > 100s of nodes!

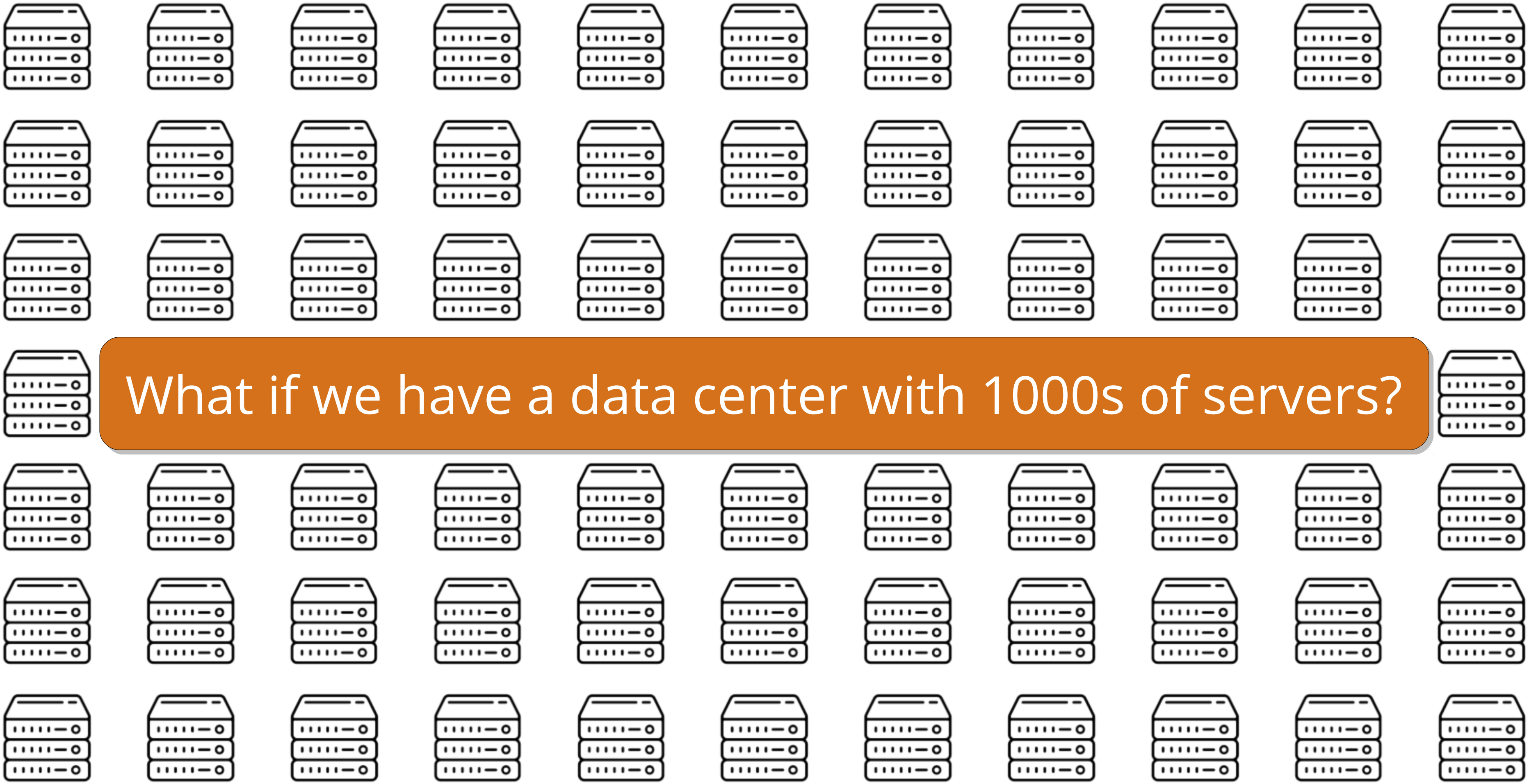


openstack

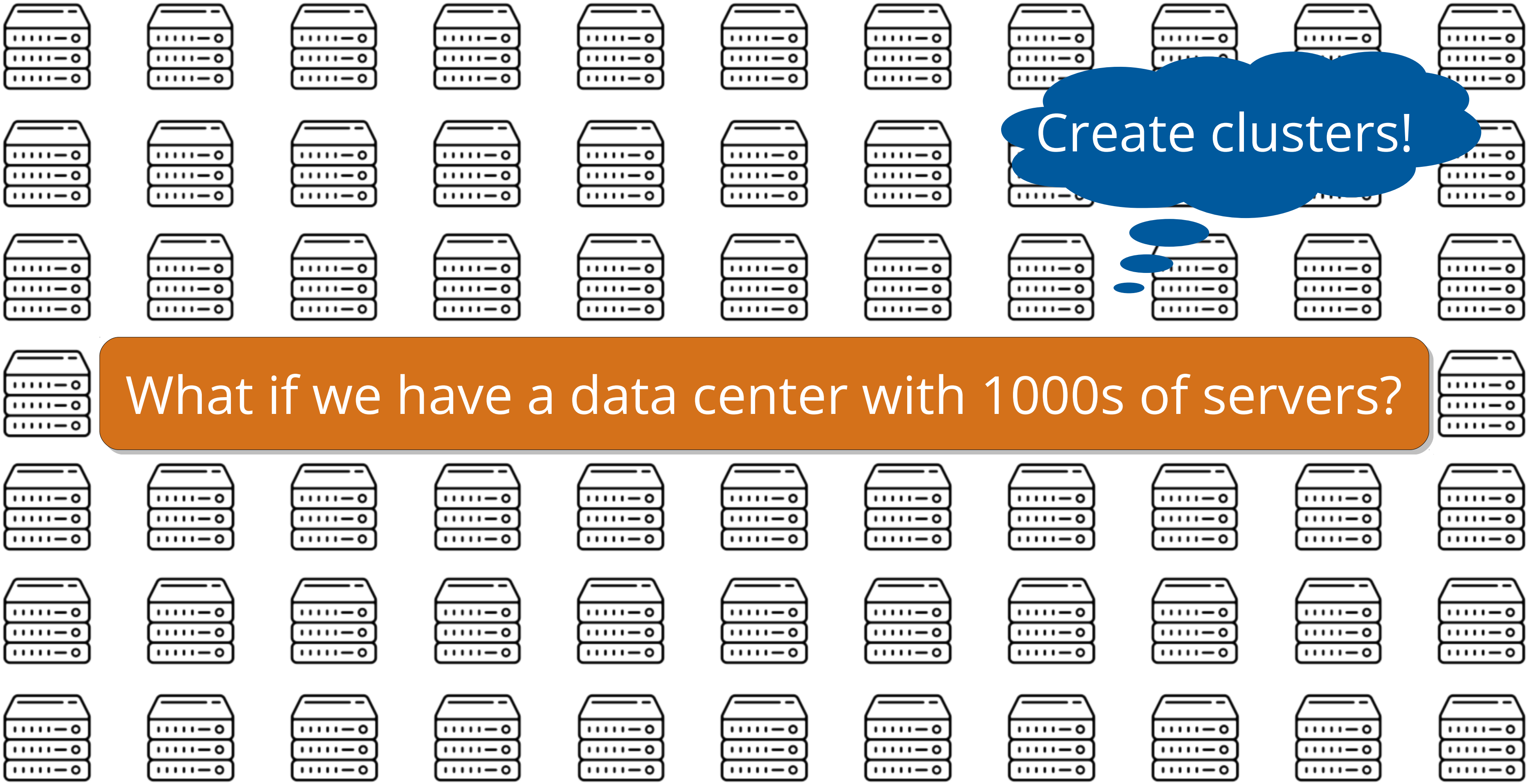
Hard to scale > 100s of nodes!



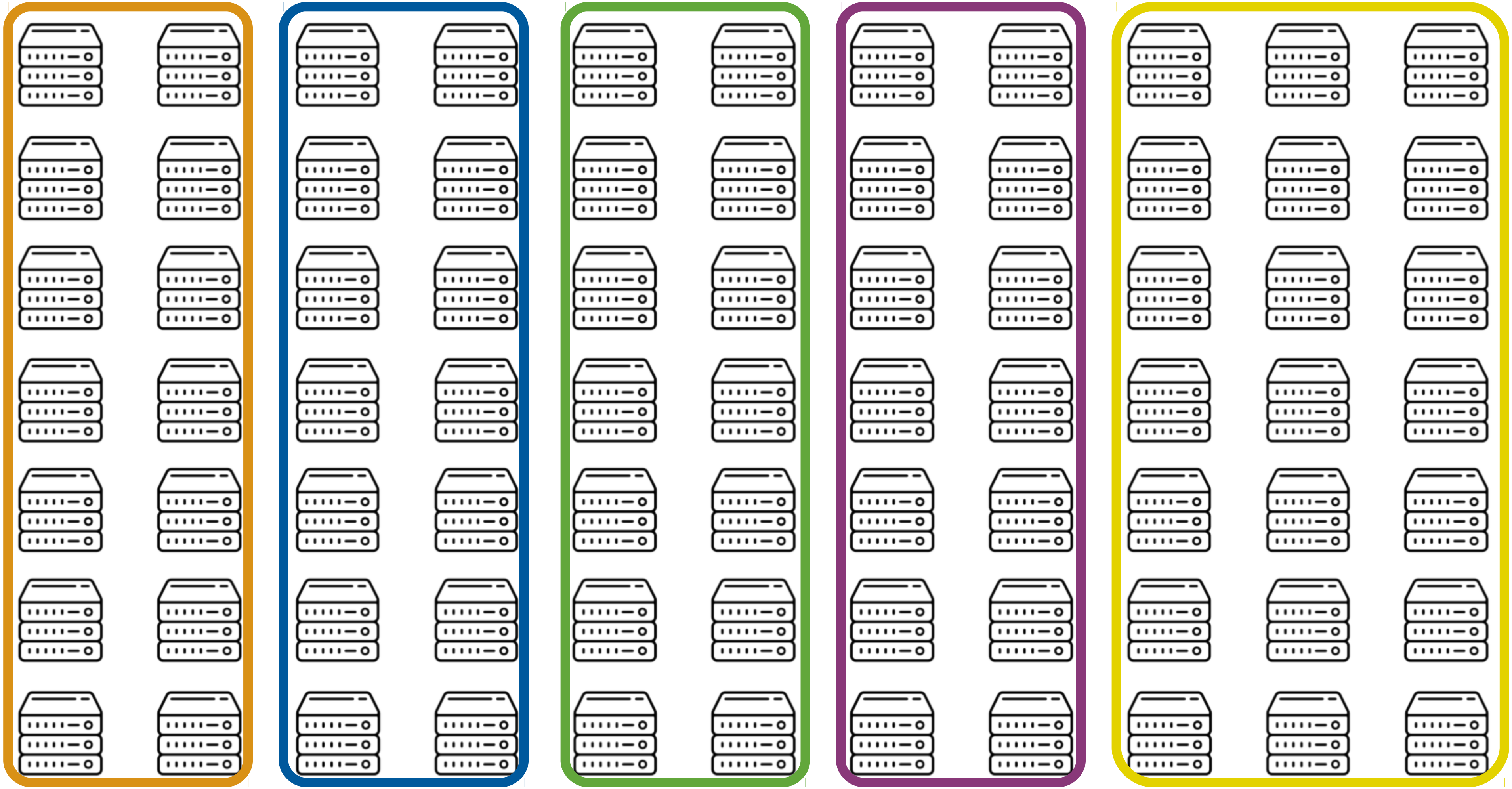
openstack™



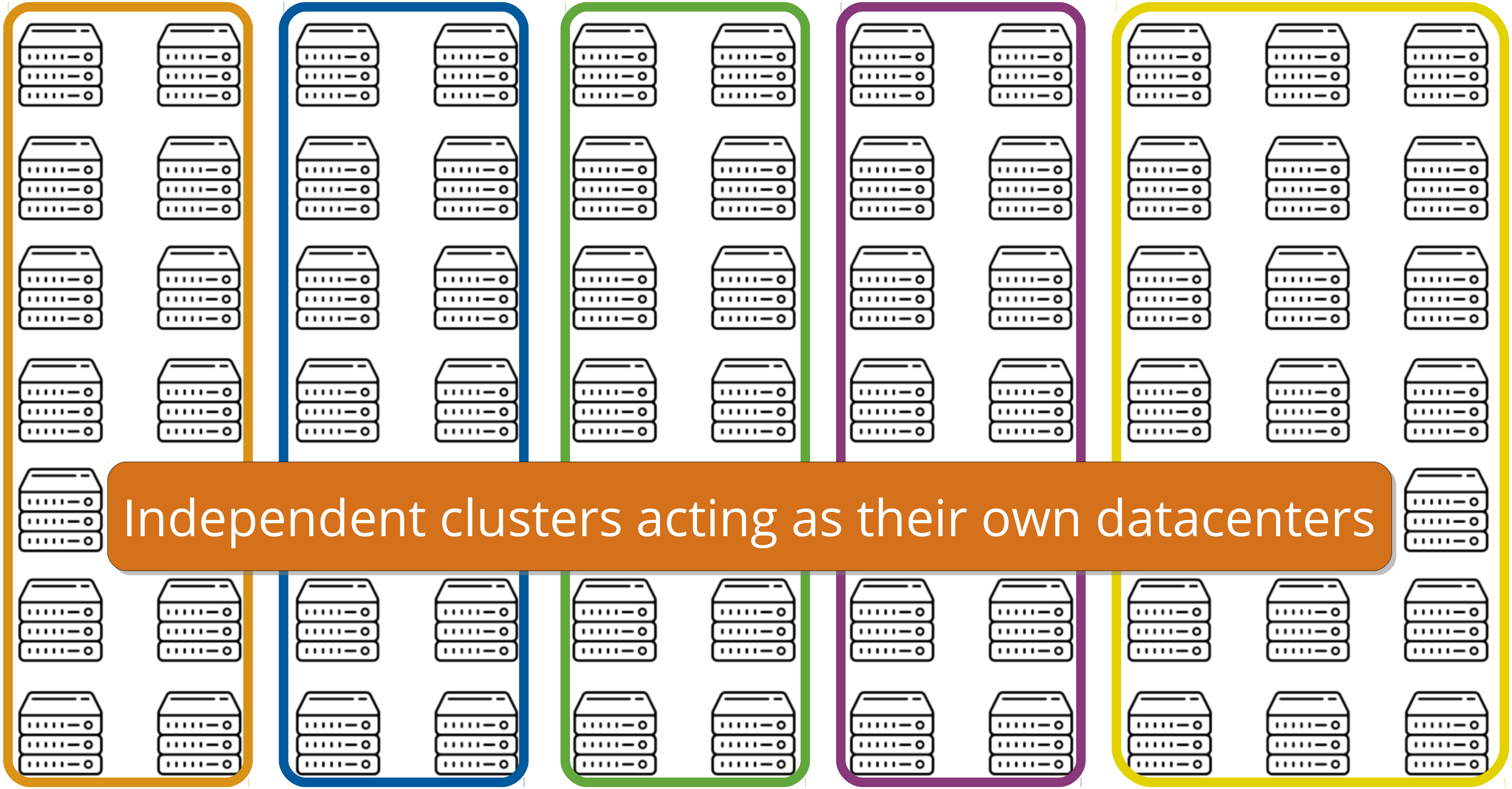
openstack™

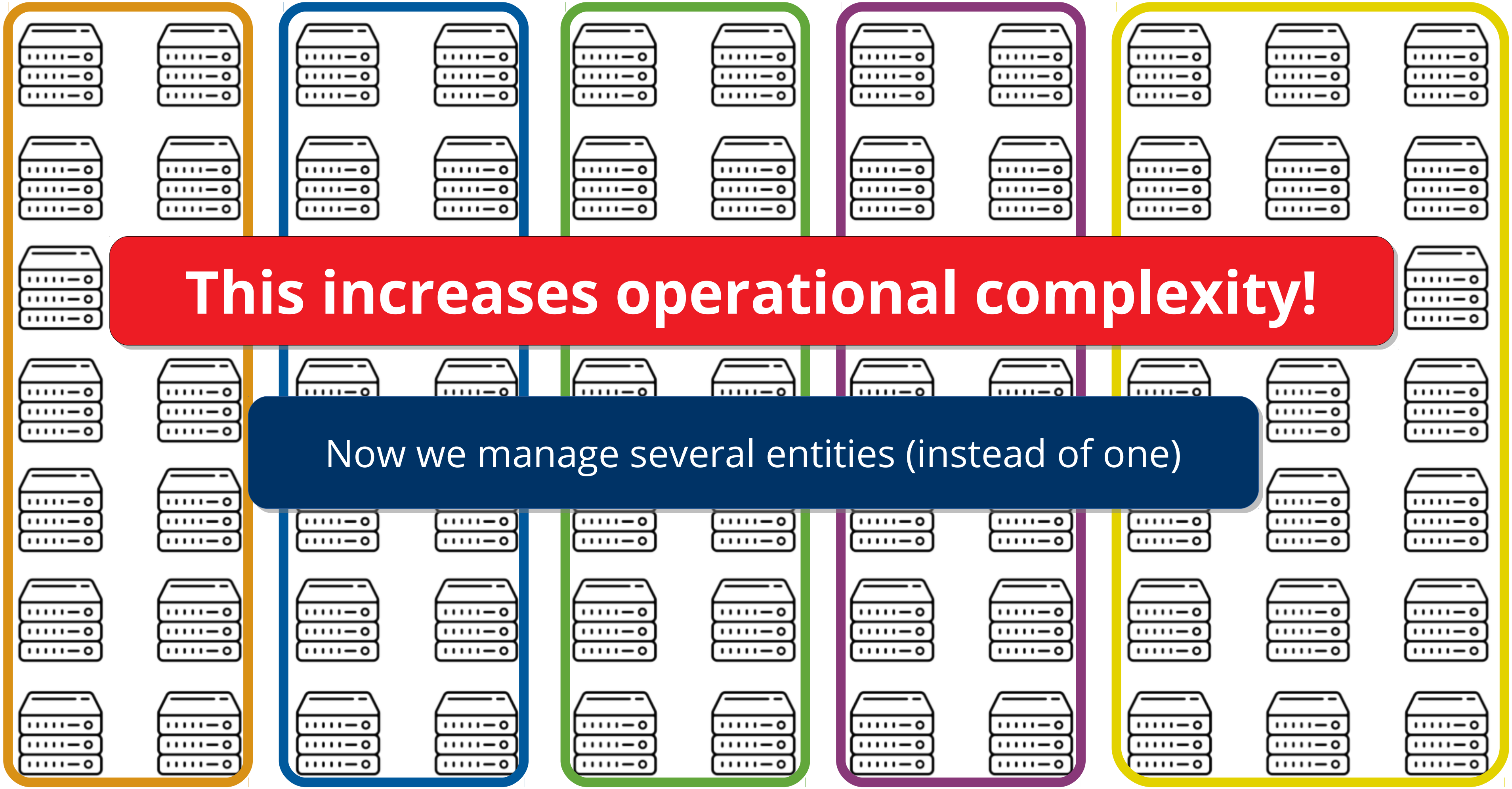

Create clusters!
What if we have a data center with 1000s of servers?

openstack™



openstack™

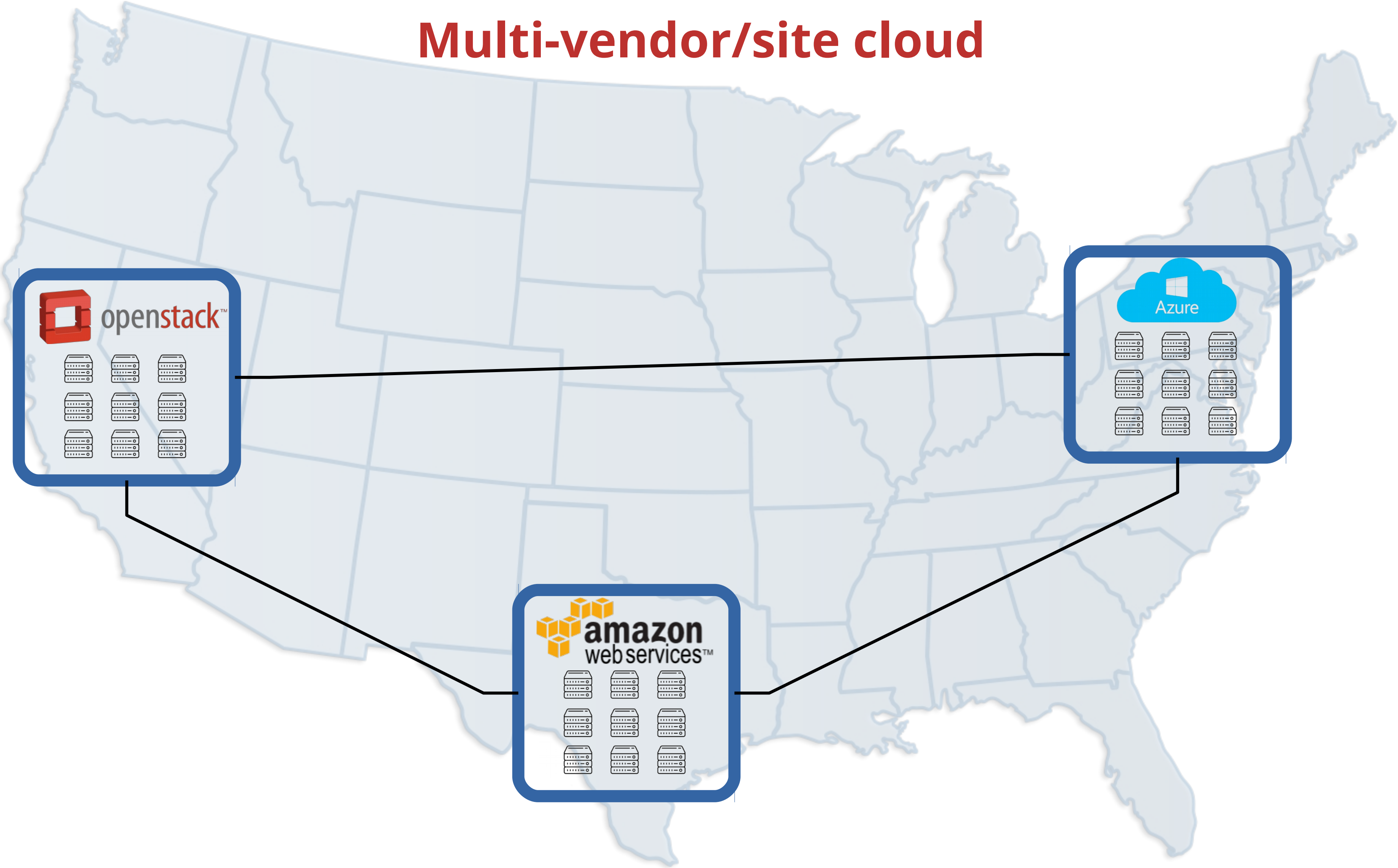




This increases operational complexity!

Now we manage several entities (instead of one)

Multi-vendor/site cloud



Multi-vendor/site cloud

Finding nodes across cloud sites/vendors is even harder!

FOCUS

Scalable and generic search service for distributed systems

Main Components

Main Components

Query Processing with Directed Pulling

FOCUS

Main Components

Query Processing with Directed Pulling

Gossip-based Node Coordination

FOCUS

Main Components

- Query Processing with Directed Pulling
- Gossip-based Node Coordination
- Easy-to-integrate Query Interface

FOCUS

Main Components

- Query Processing with Directed Pulling
- Gossip-based Node Coordination
- Easy-to-integrate Query Interface

FOCUS

Abstractions

Abstractions

Node Attributes

Abstractions

Node Attributes

- Static
- Dynamic

Abstractions

Node Attributes

- Static
  - Never change
- Dynamic

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic

Abstractions

- **Node Attributes**
- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Query Structure

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Query Structure

- Attribute List

Abstractions

Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Query Structure

- Attribute List
  - *name (string)*
  - *upper bound (int)*
  - *lower bound (int)*

Abstractions

Node Attributes

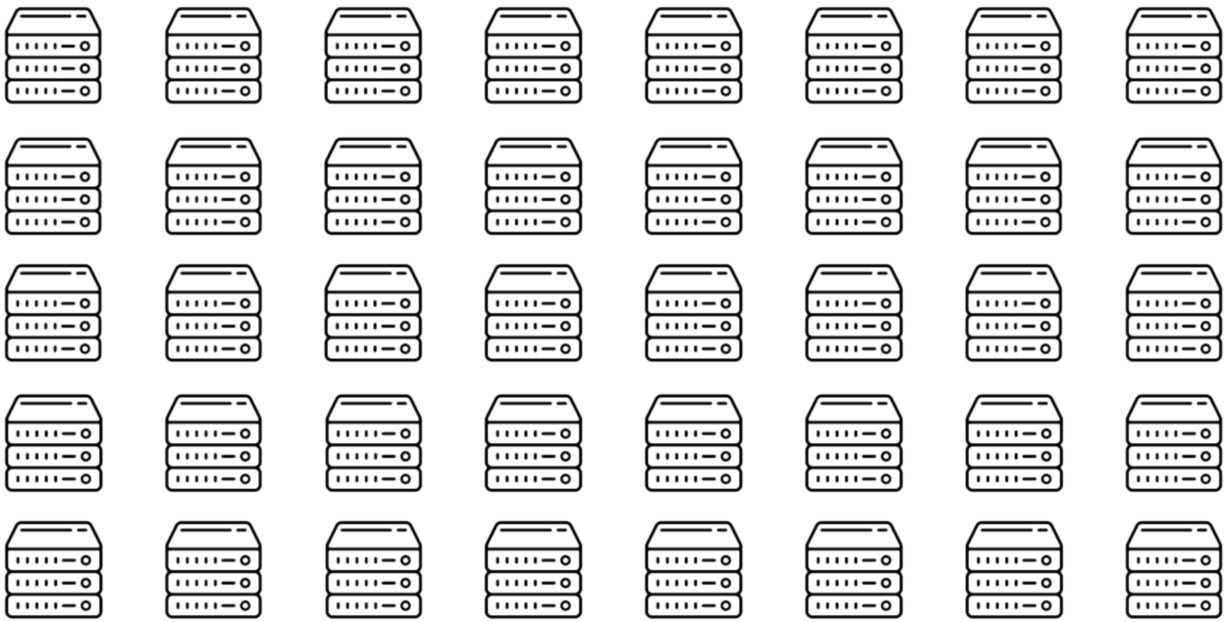
- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

Query Structure

- Attribute List
  - *name* *(string)*
  - *upper bound* *(int)*
  - *lower bound* *(int)*
- *limit* *(int)*
- *freshness* *(int)*

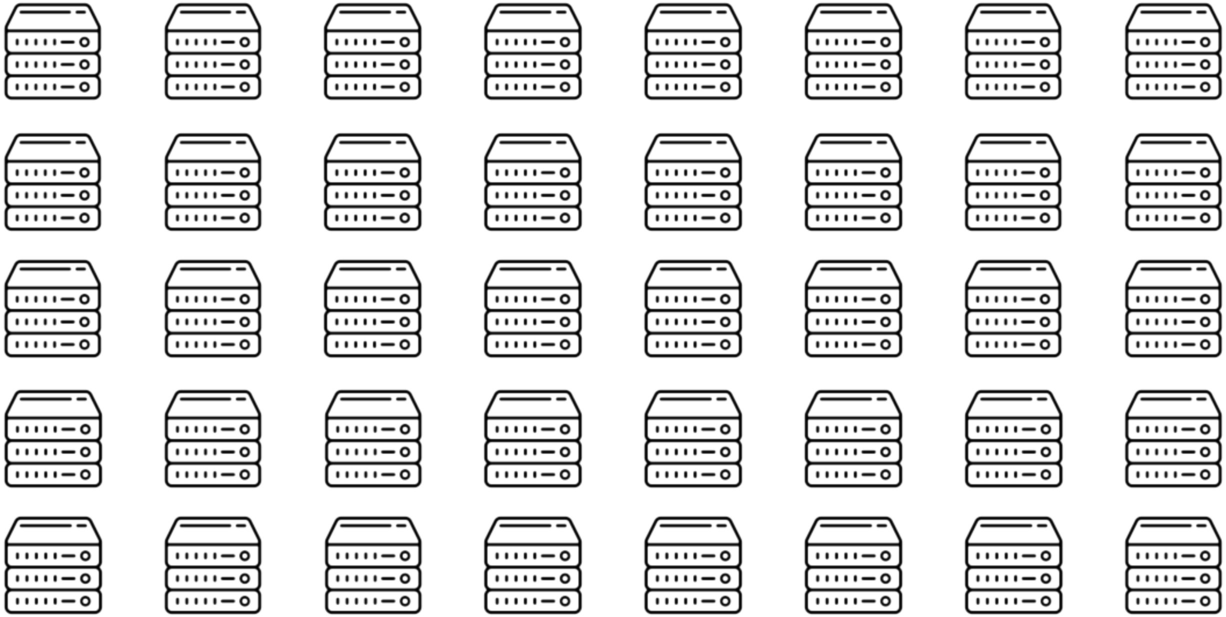
Query Processing with Directed Pulling

Attribute-based Grouping



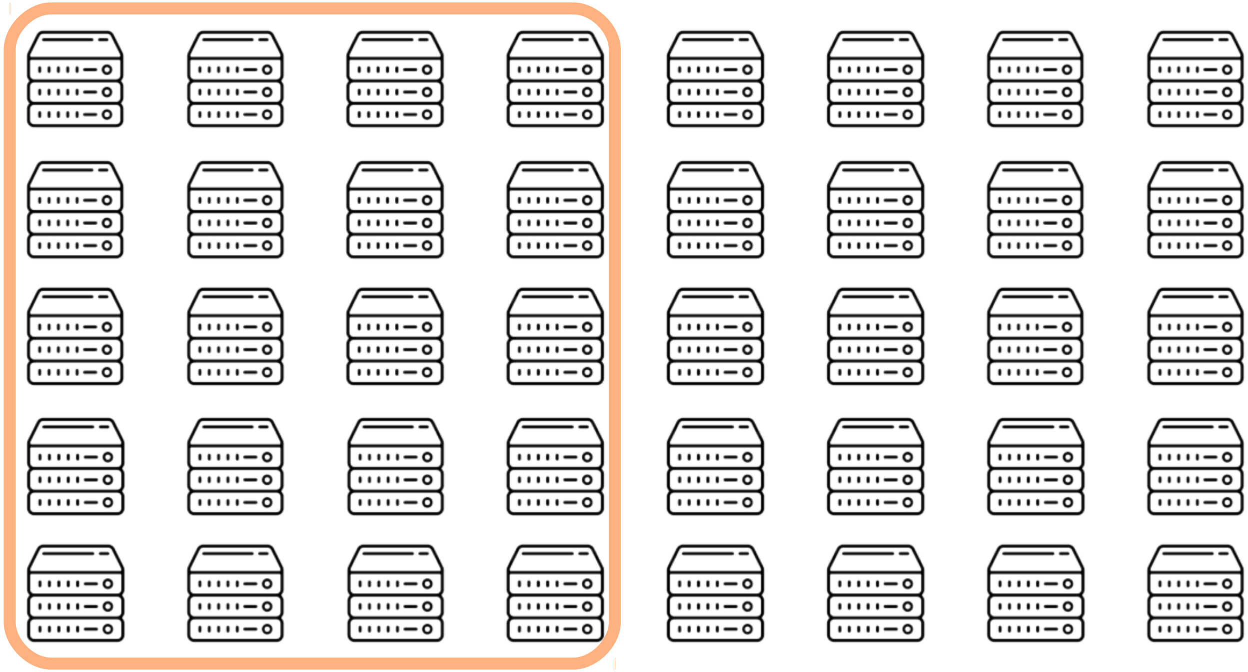
Attribute-based Grouping

Group nodes according to their attribute values



Attribute-based Grouping

cpu_usage {50-100}%



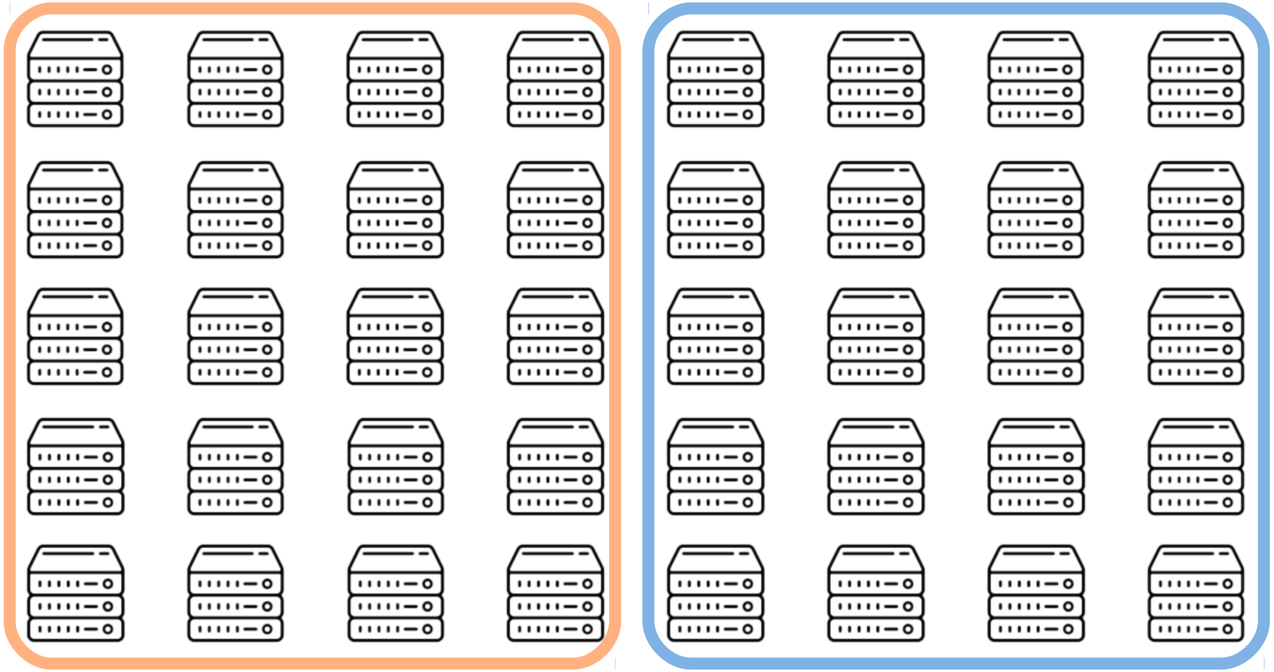
Group nodes according to their attribute values

Attribute-based Grouping

cpu_usage {50-100}%

cpu_usage {0-50}%

Group nodes according to their attribute values



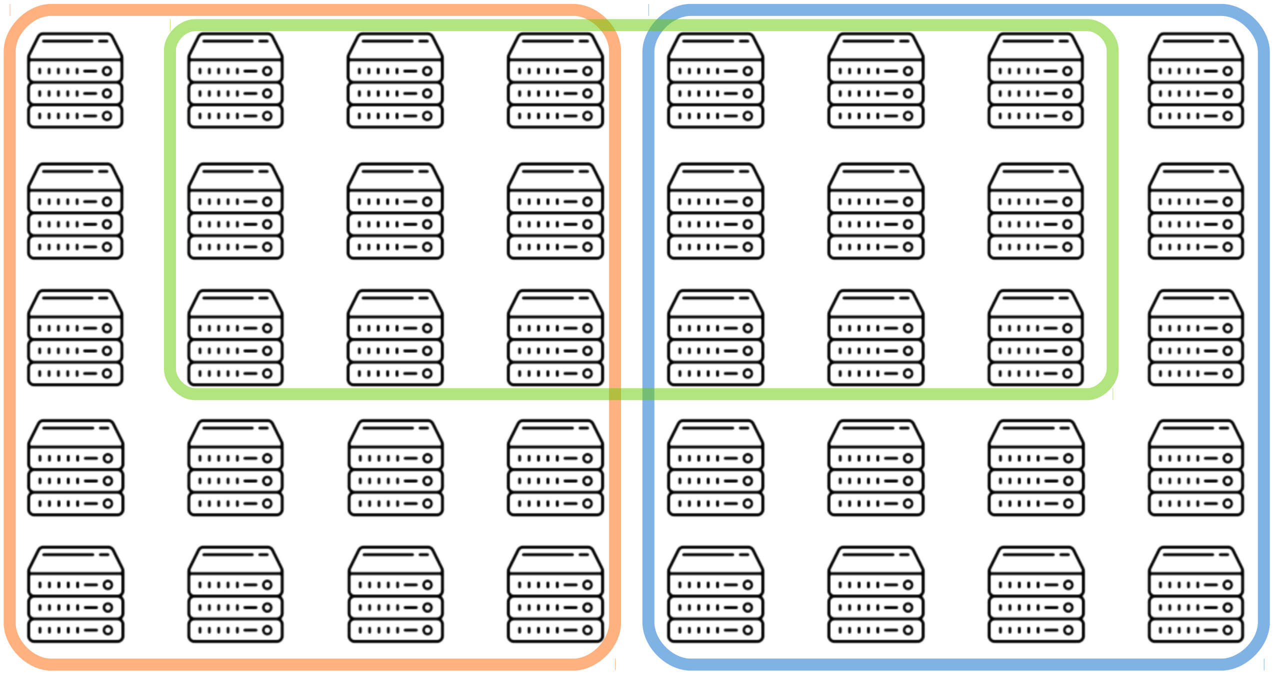
Attribute-based Grouping

cpu_usage {50-100}%

cpu_usage {0-50}%

avail_RAM {4-8}GB

Group nodes according to their attribute values



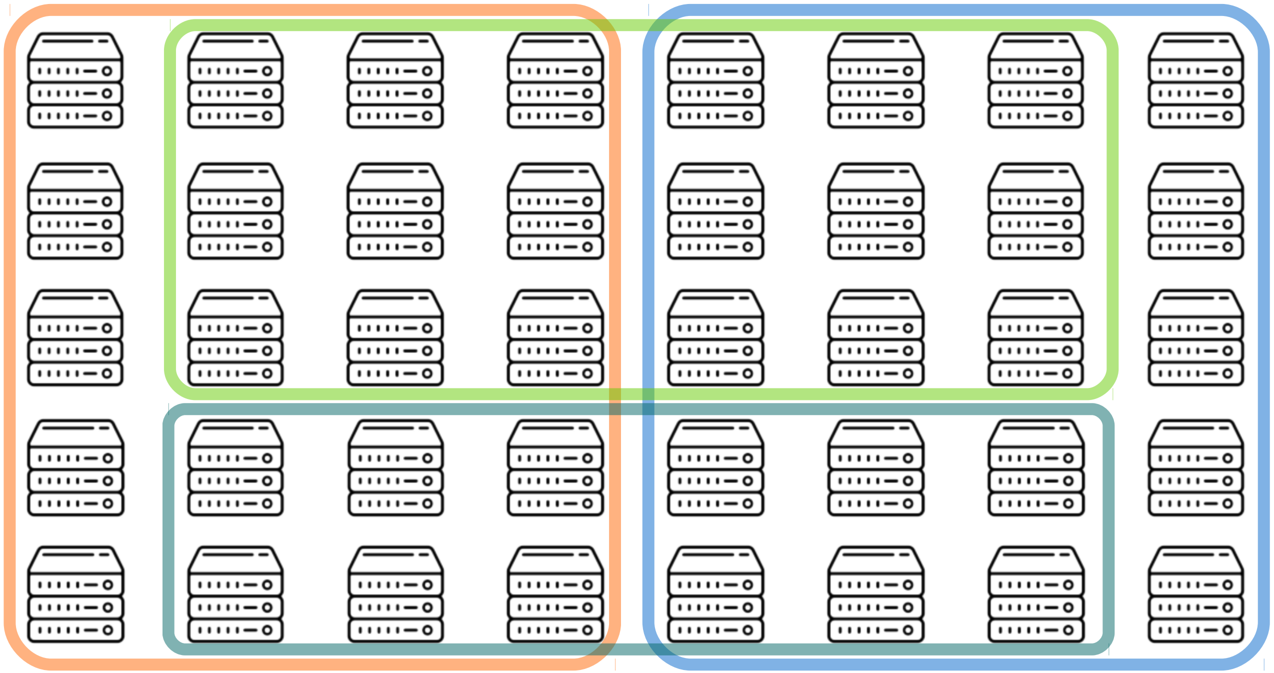
Attribute-based Grouping

cpu_usage {50-100}%

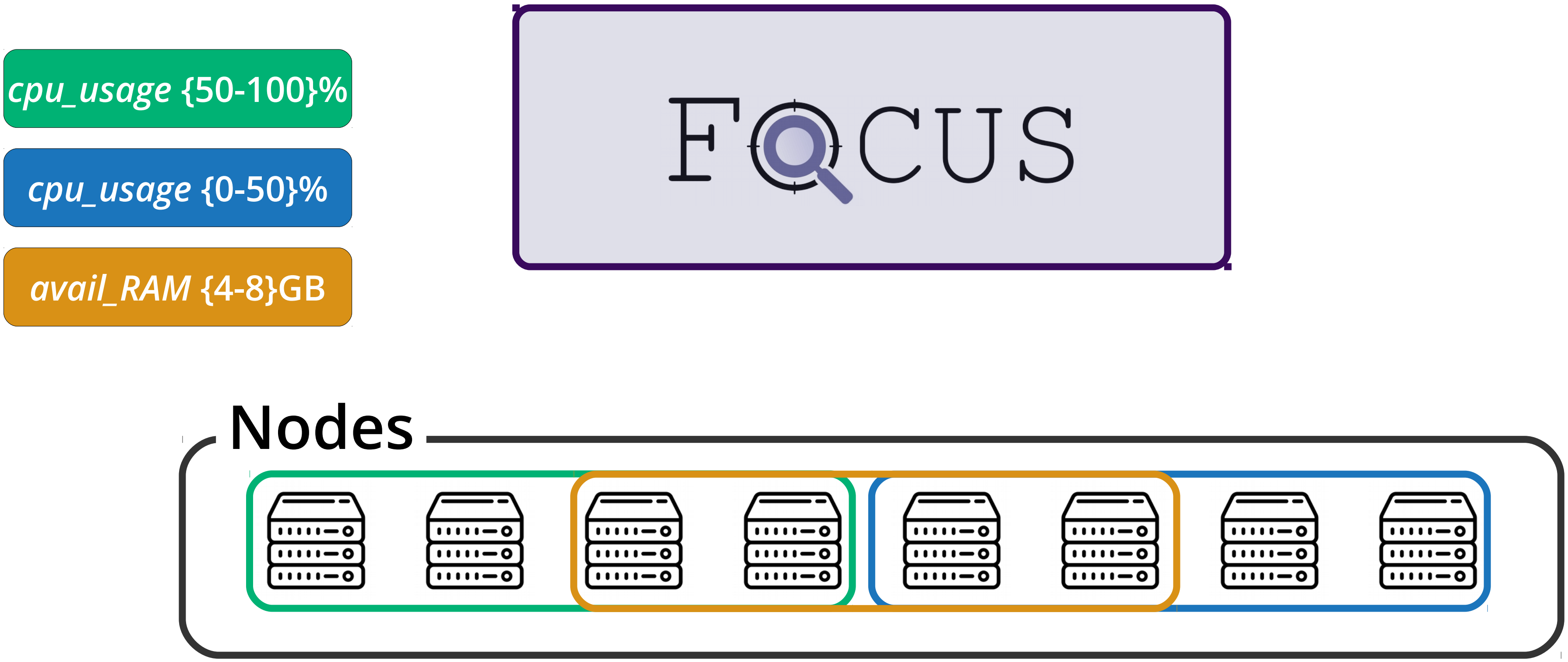
cpu_usage {0-50}%

avail_RAM {4-8}GB

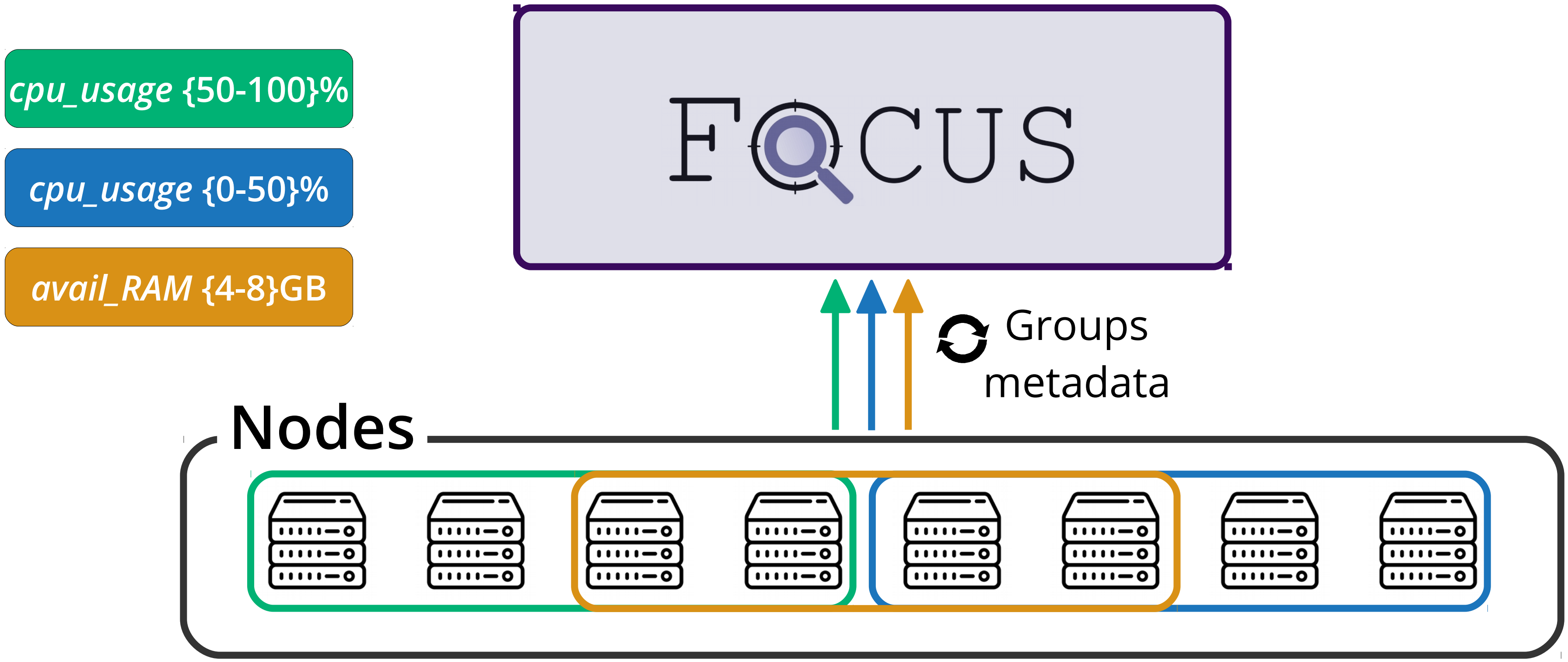
cpu_cores {8-12}



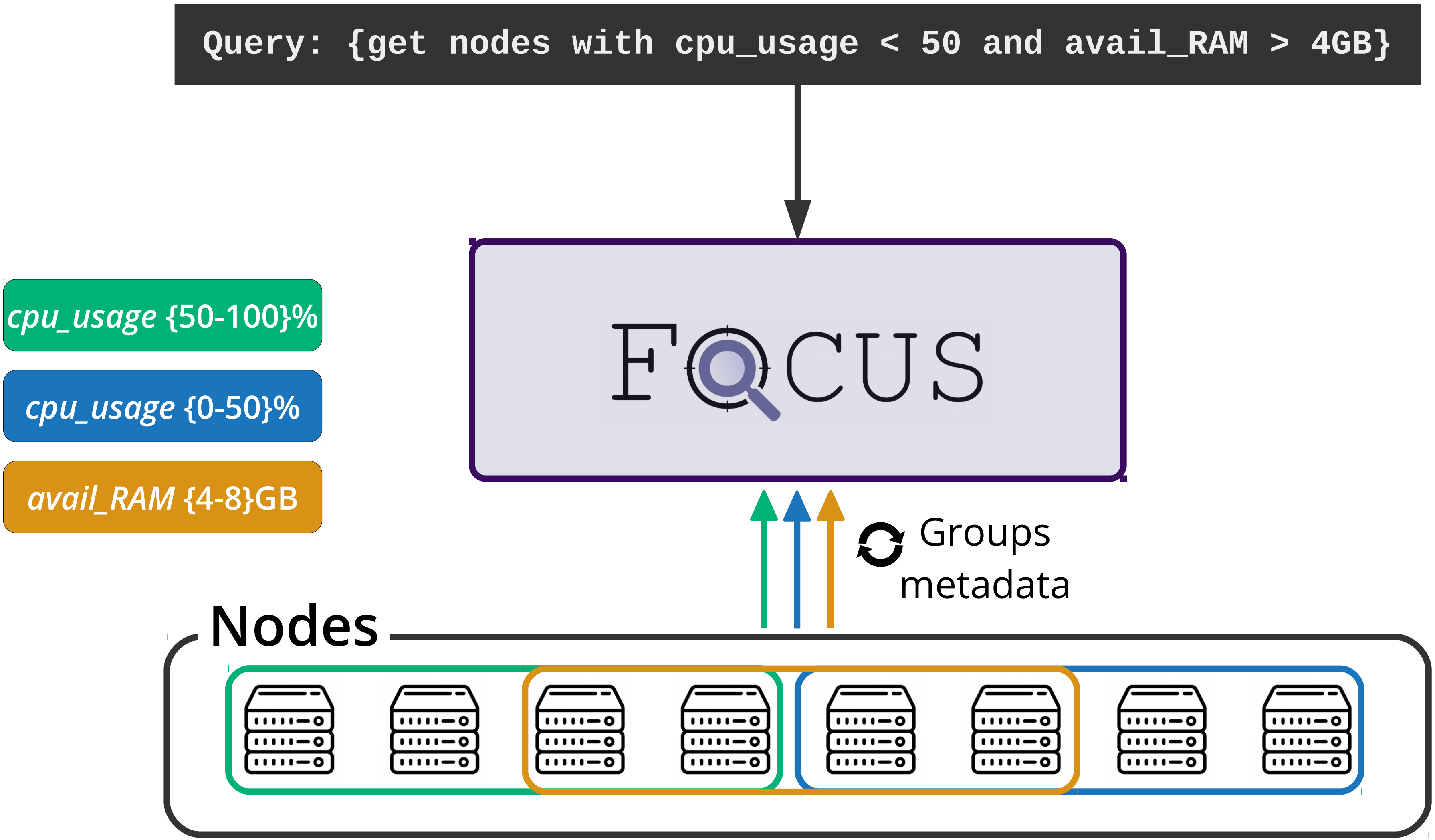
Group nodes according to their attribute values

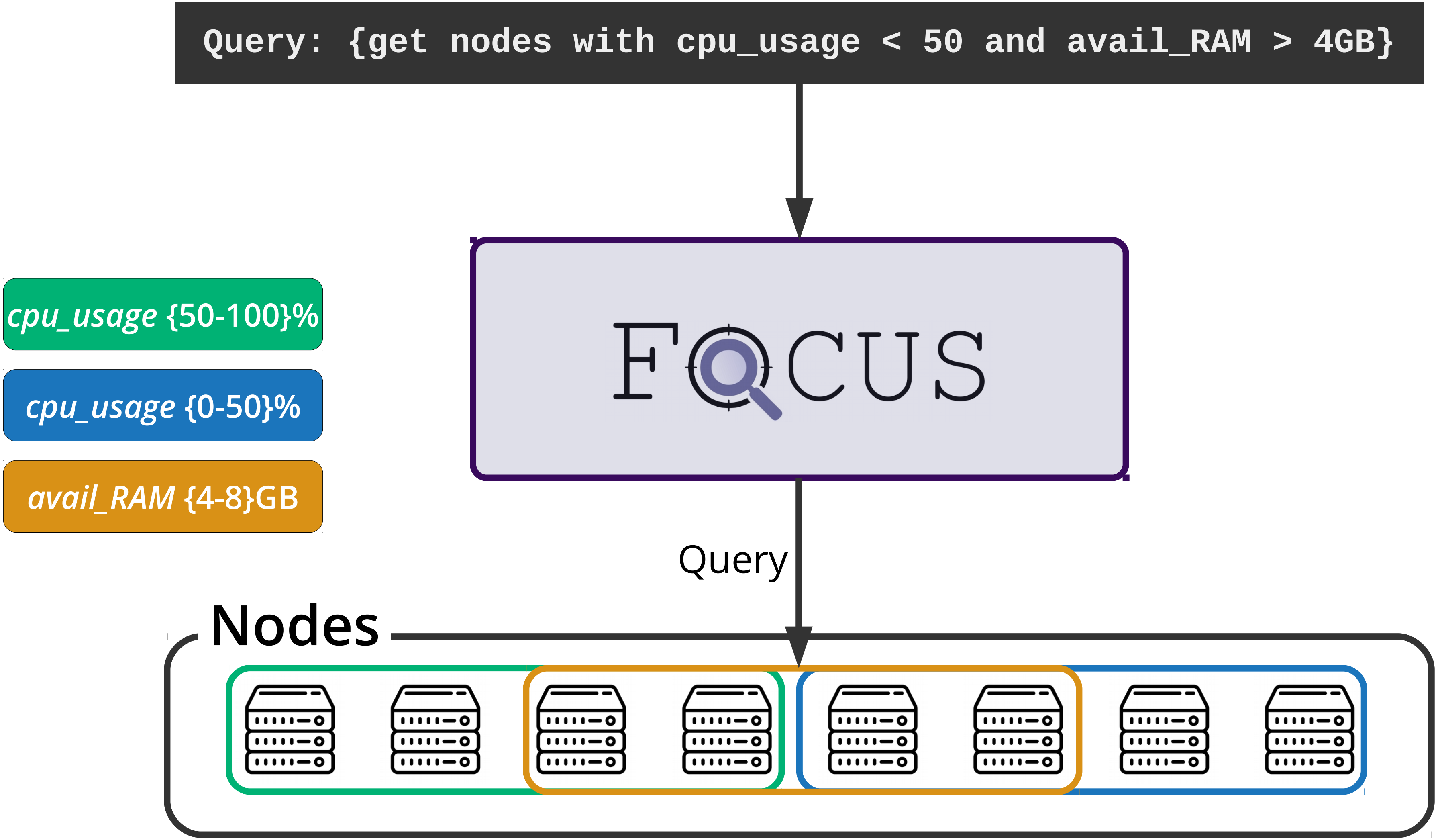


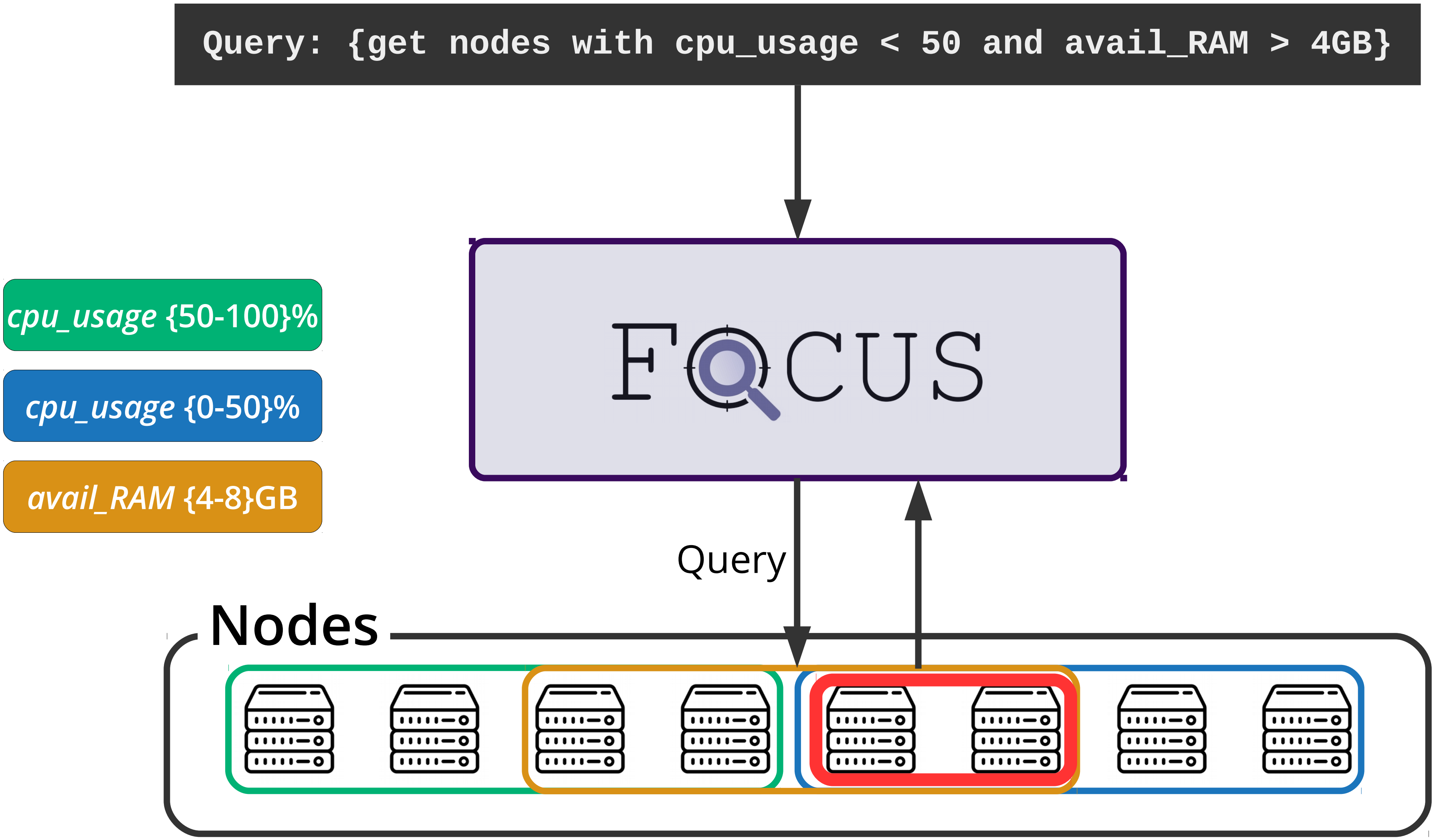
cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
FOCUS
Nodes

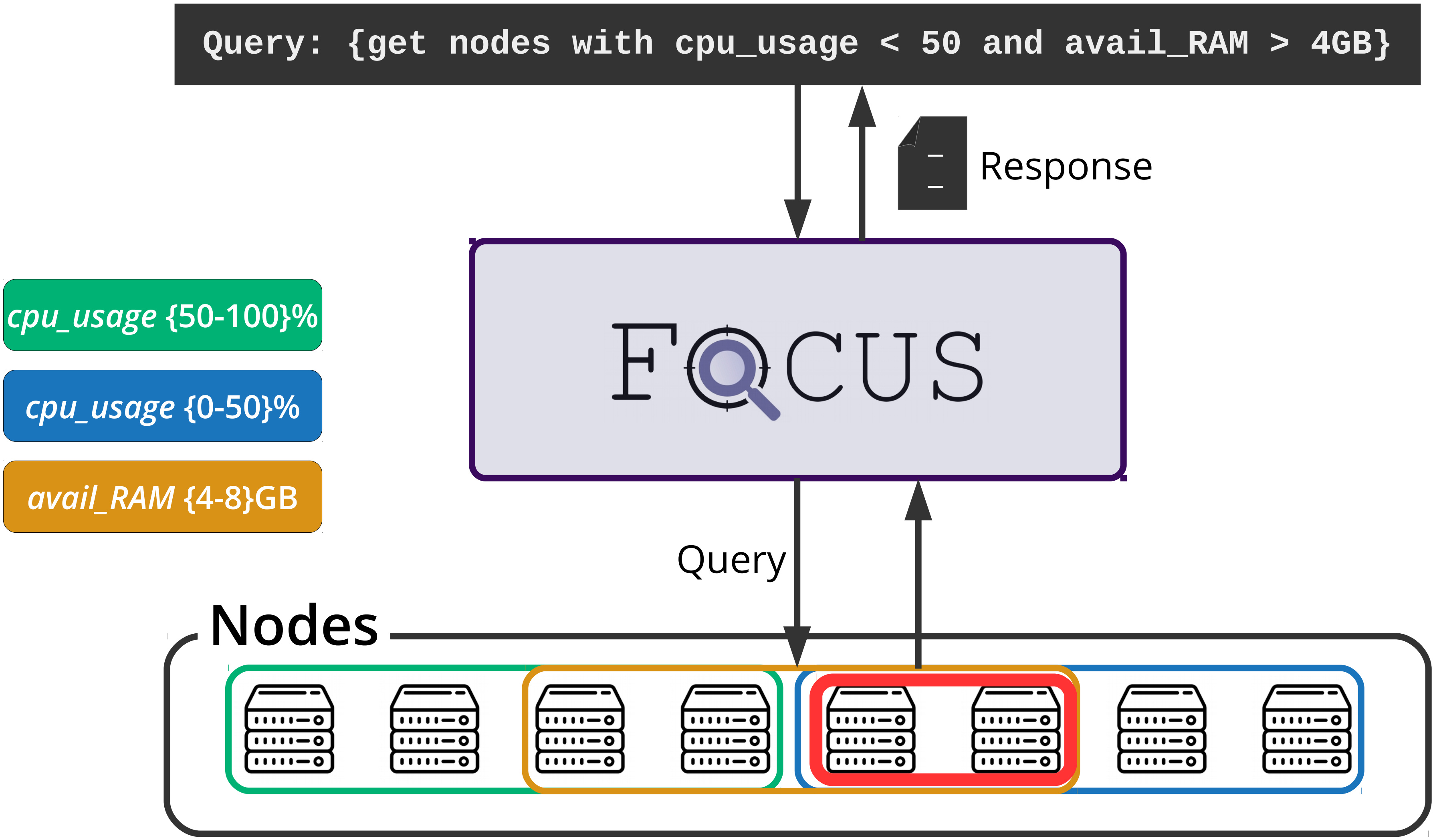


cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
FOCUS
Groups metadata
Nodes


Query: {get nodes with cpu_usage < 50 and avail_RAM > 4GB}
cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
FOCUS
Groups metadata
Nodes


Query: {get nodes with cpu_usage < 50 and avail_RAM > 4GB}
FOCUS
cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
Query
Nodes

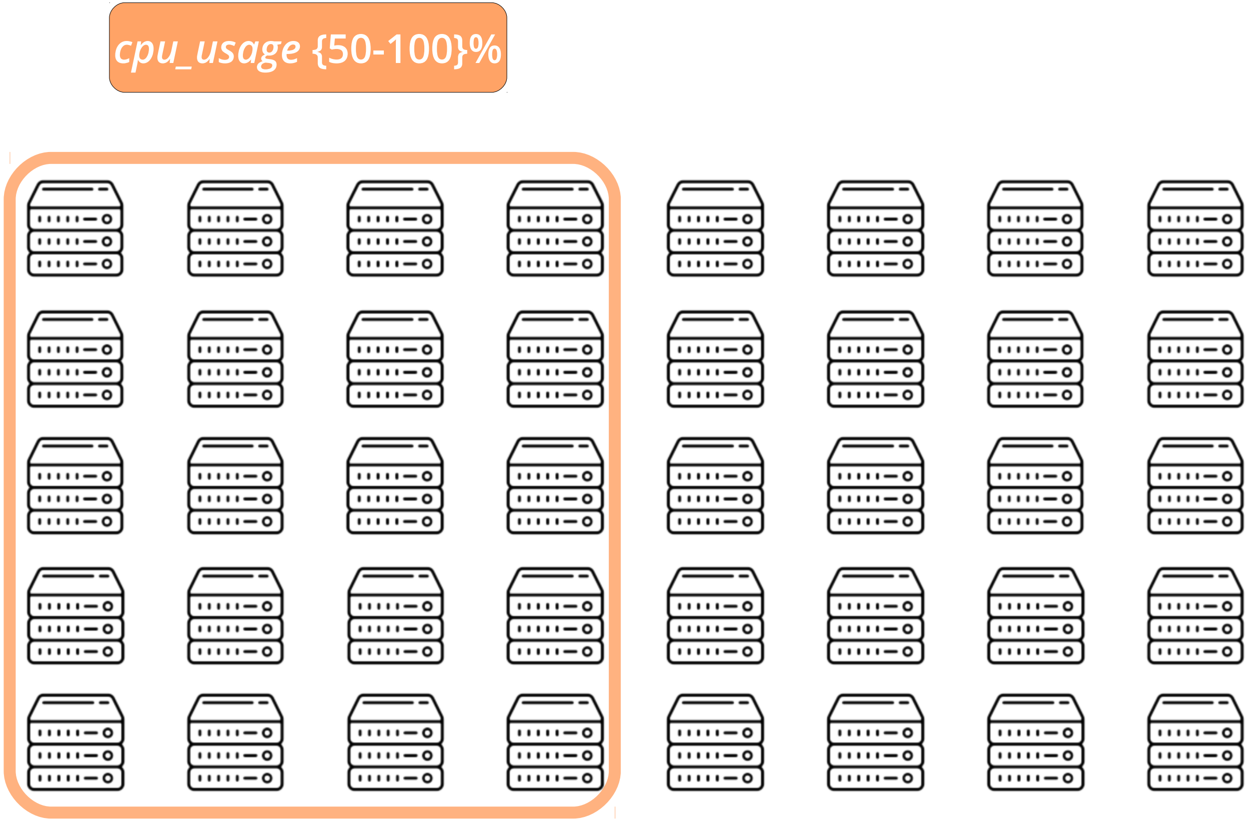

Query: {get nodes with cpu_usage < 50 and avail_RAM > 4GB}
cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
FOCUS
Query
Nodes



Query: {get nodes with cpu_usage < 50 and avail_RAM > 4GB}
Response
cpu_usage {50-100}%
cpu_usage {0-50}%
avail_RAM {4-8}GB
FOCUS
Query
Nodes

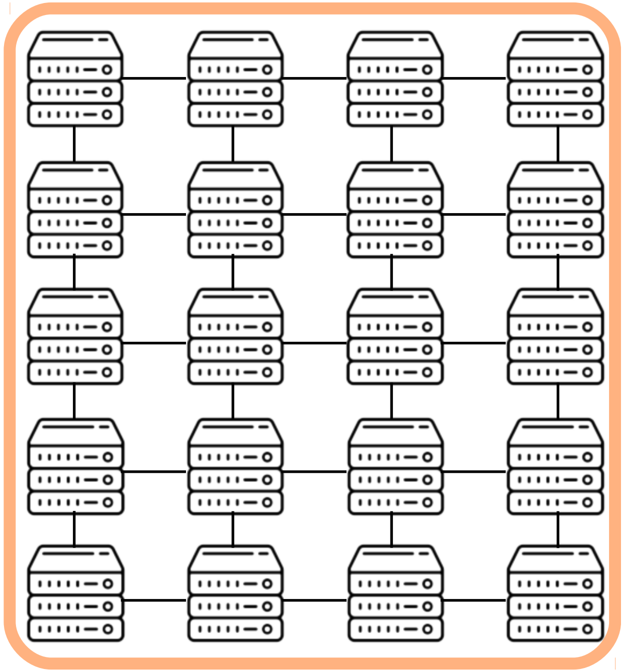
Gossip-based Node Coordination

Gossip-based Coordination



Gossip-based Coordination

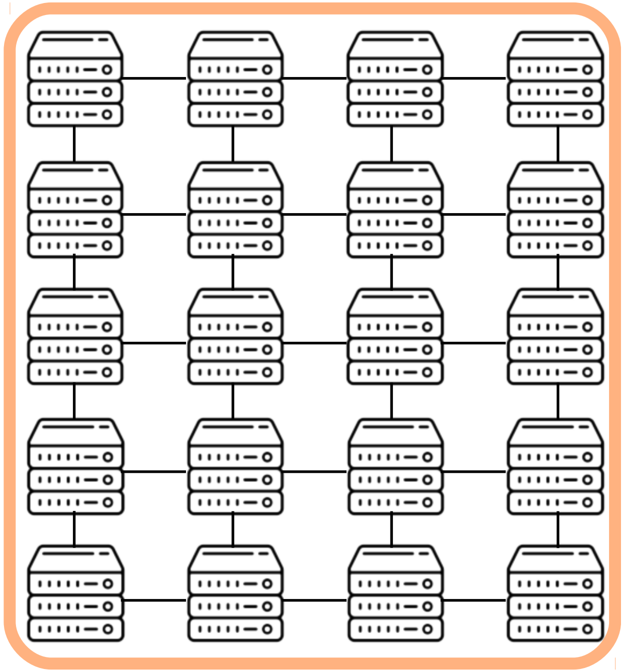
cpu_usage {50-100}%



Nodes in a group are connected through a **p2p *gossip*** channel

Gossip-based Coordination

cpu_usage {50-100}%

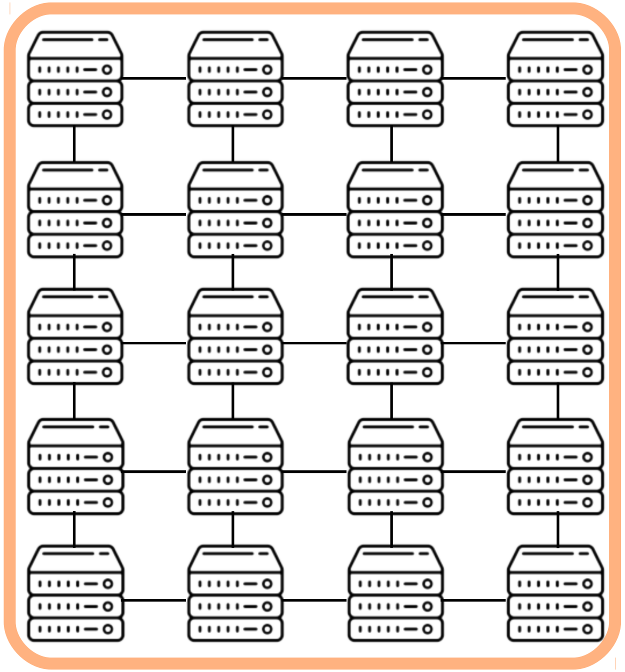


Nodes in a group are connected through a **p2p *gossip*** channel

Nodes exchange membership information

Gossip-based Coordination

cpu_usage {50-100}%



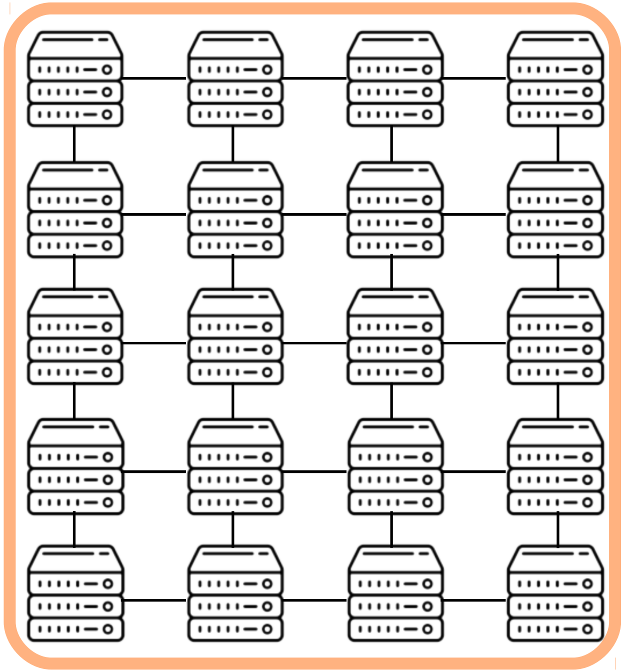
Nodes in a group are connected through a **p2p *gossip*** channel

Nodes exchange membership information

One node pushes group info to the FOCUS server

Gossip-based Coordination

cpu_usage {50-100}%



Nodes in a group are connected through a **p2p *gossip*** channel

Nodes exchange membership information

One node pushes group info to the FOCUS server

Queries are propagated via *gossip* channel

Dynamic Groups Management

Operations flow in FOCUS

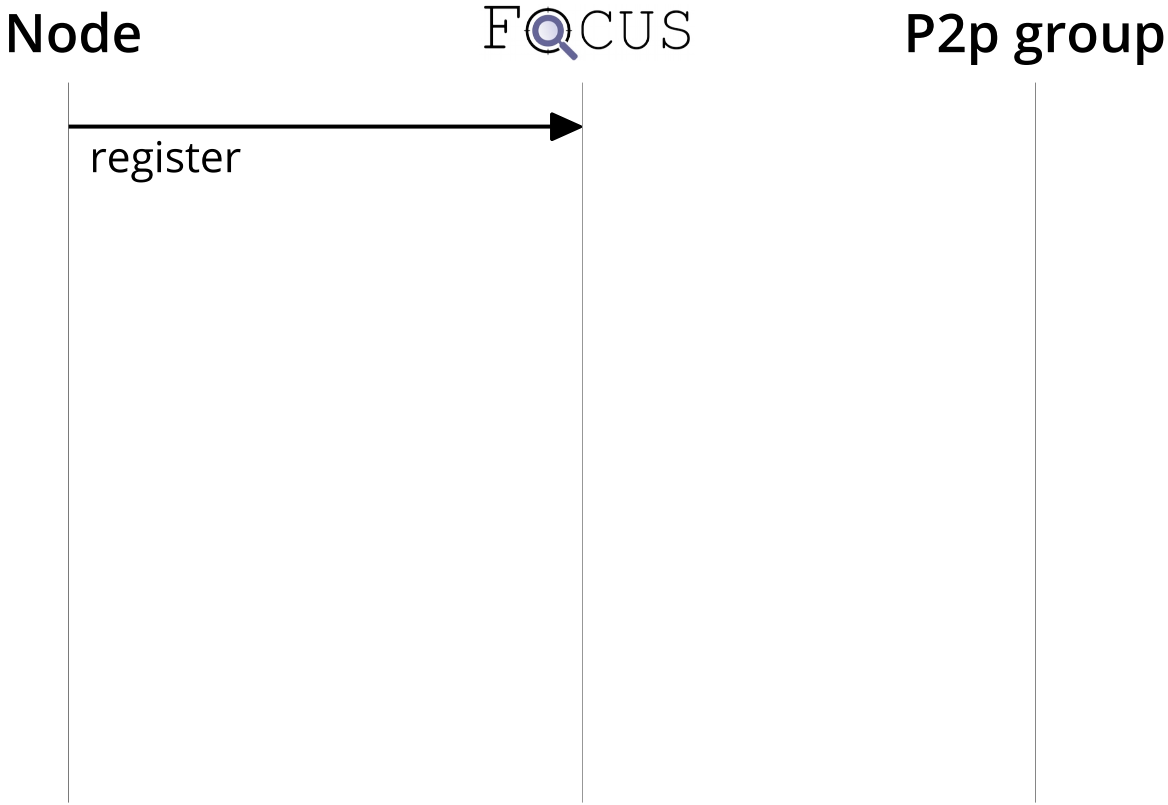
Node

FOCUS

P2p group

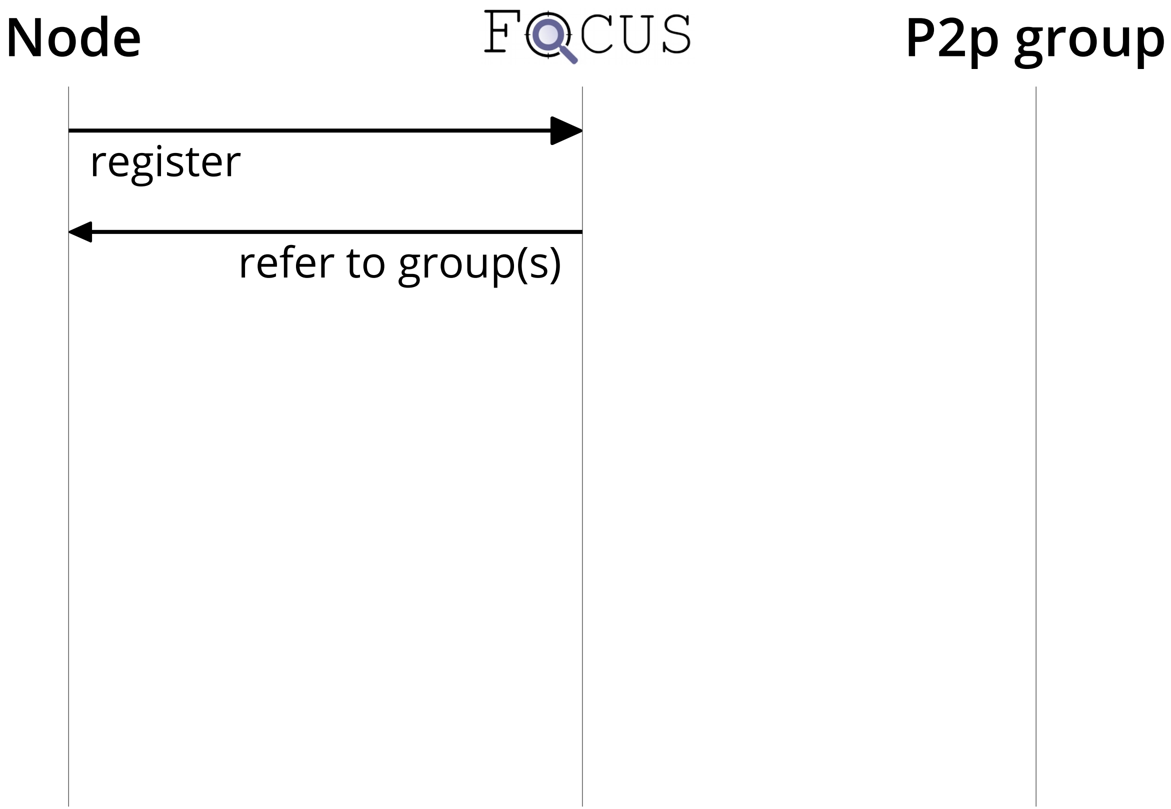
Dynamic Groups Management

Operations flow in FOCUS



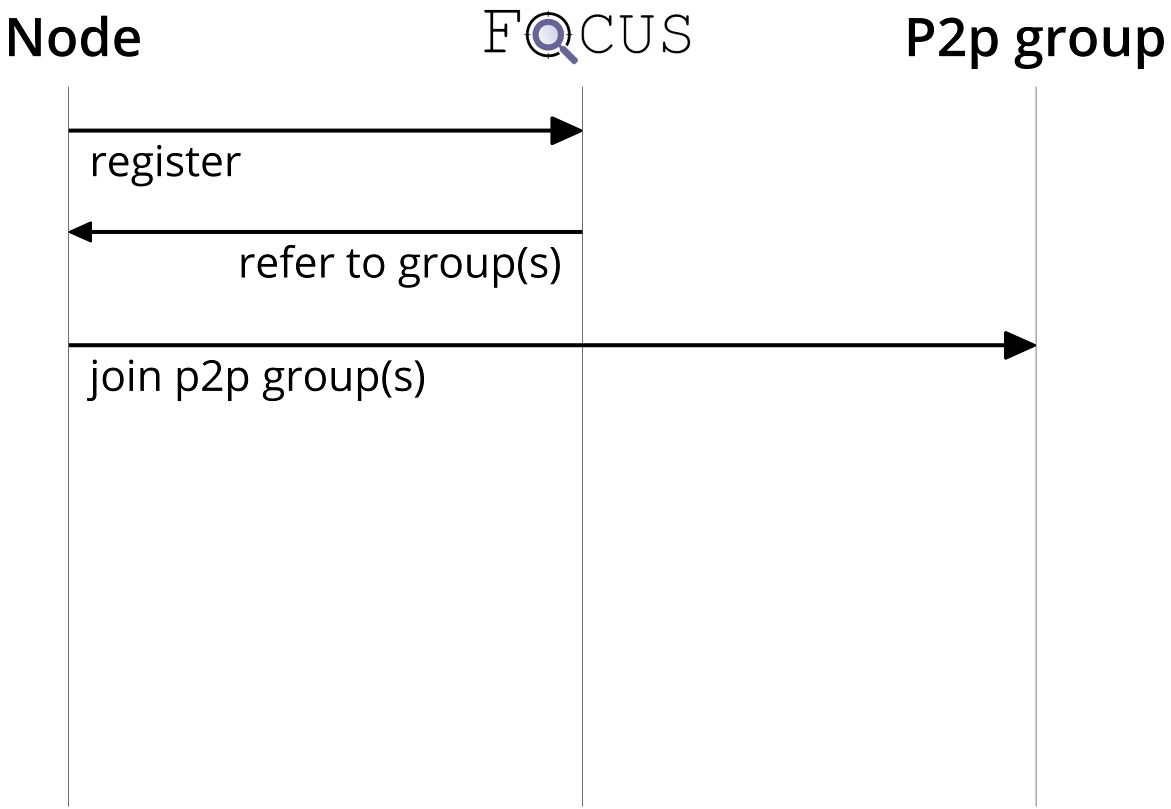
Dynamic Groups Management

Operations flow in FOCUS



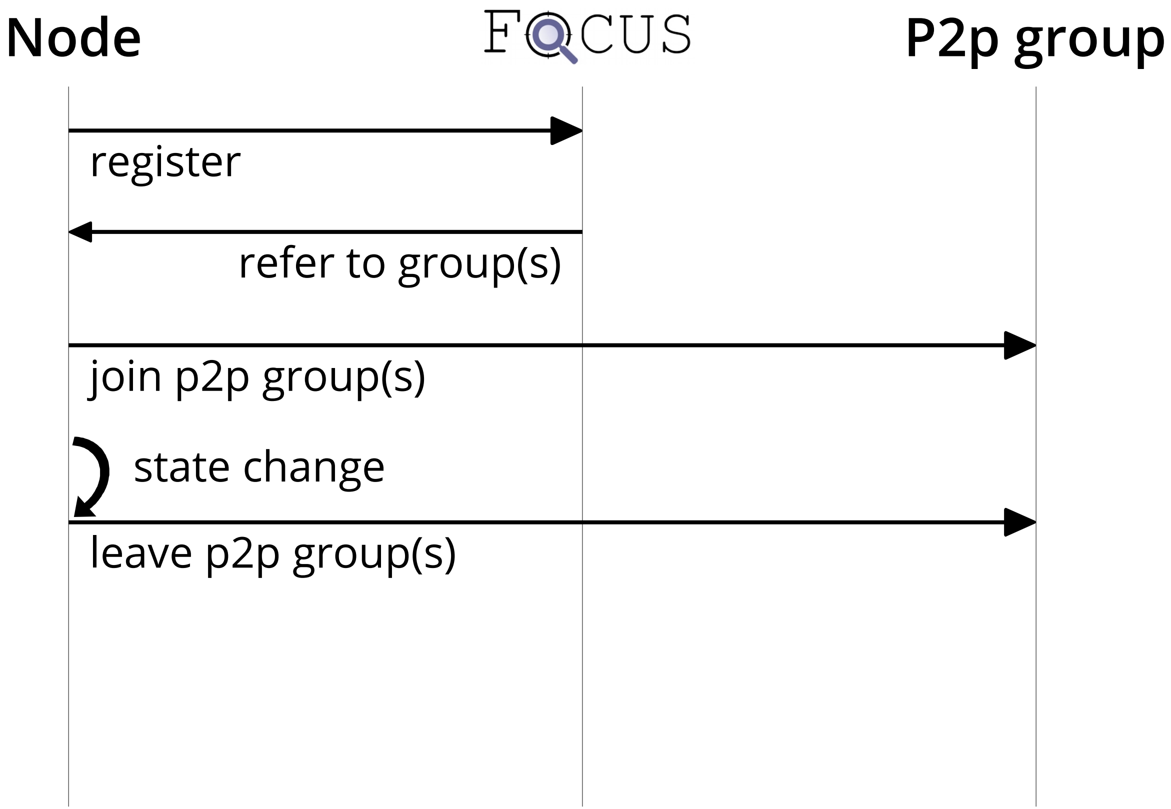
Dynamic Groups Management

Operations flow in FOCUS



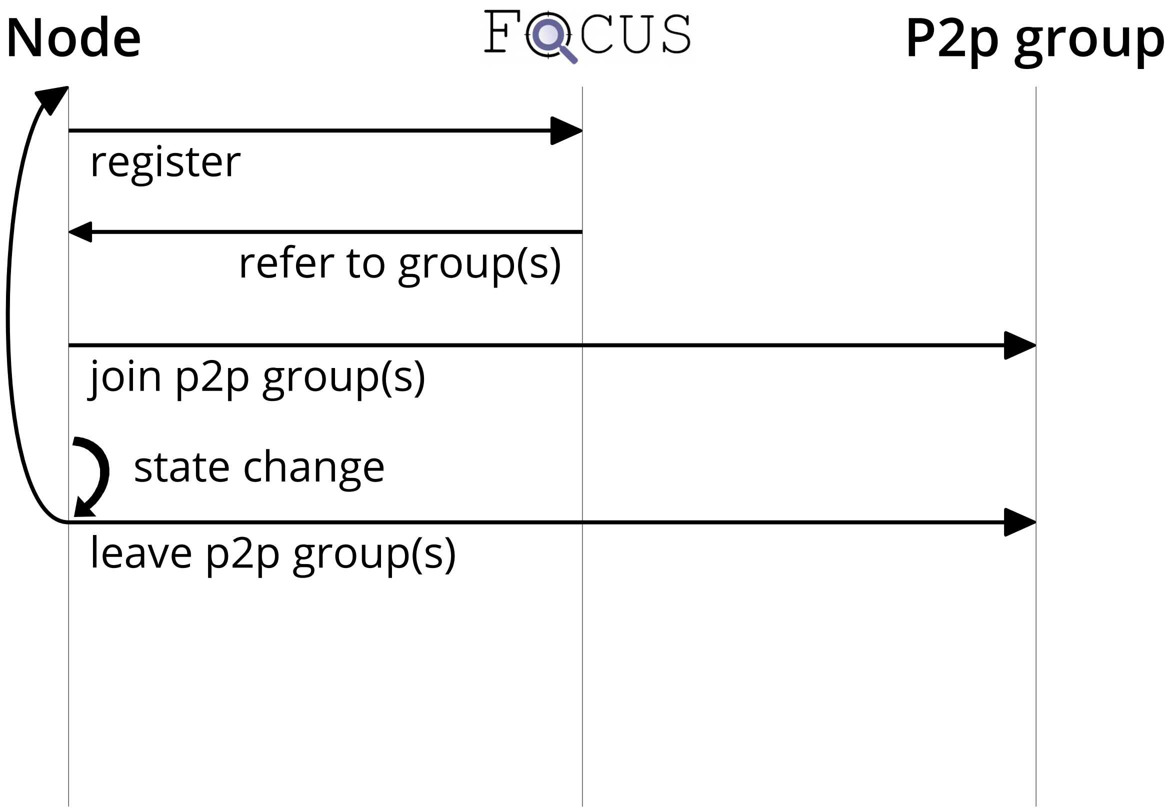
Dynamic Groups Management

Operations flow in FOCUS



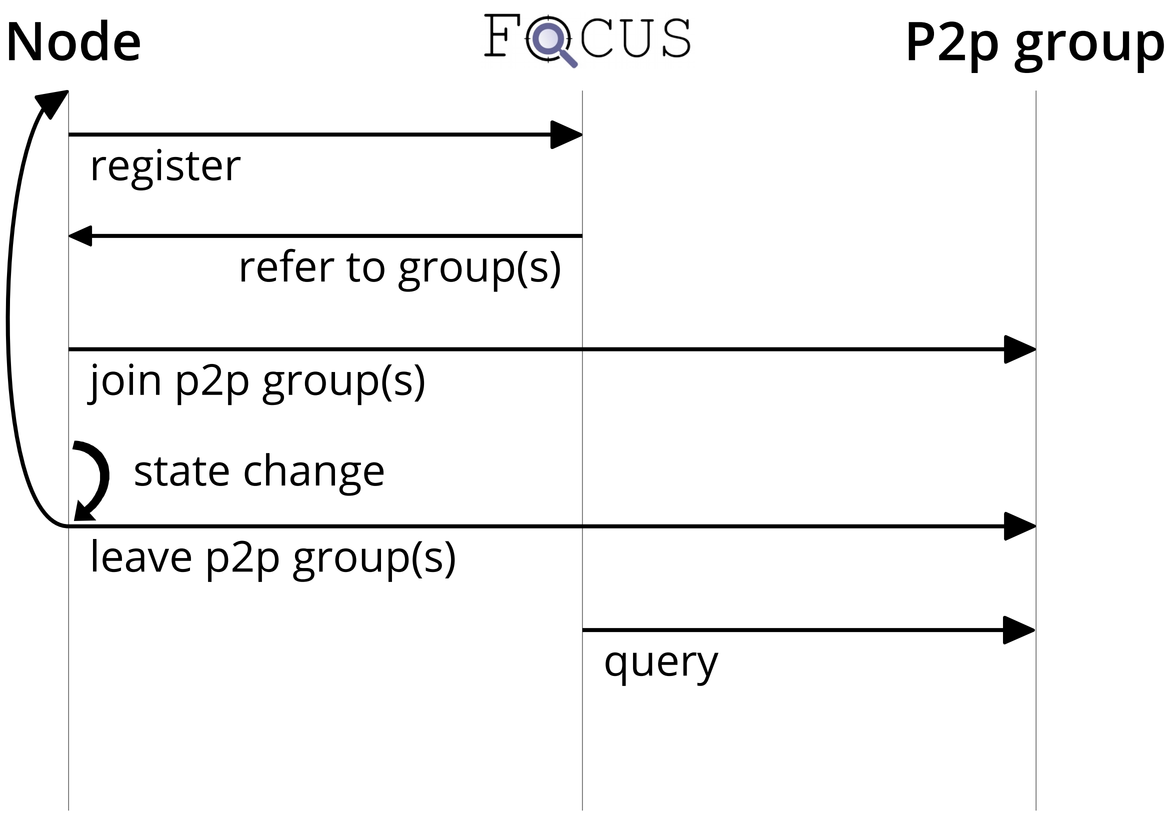
Dynamic Groups Management

Operations flow in FOCUS



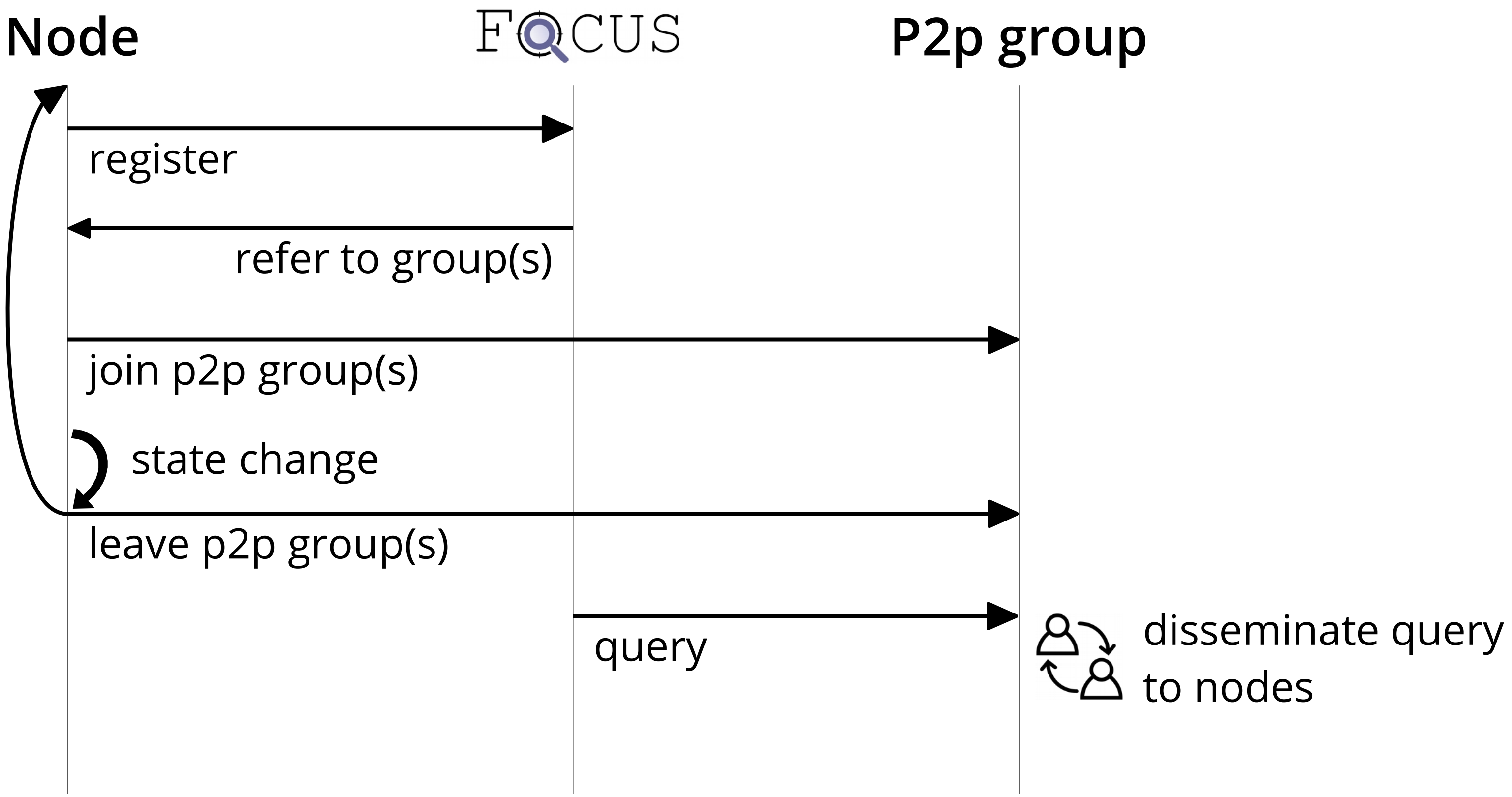
Dynamic Groups Management

Operations flow in FOCUS



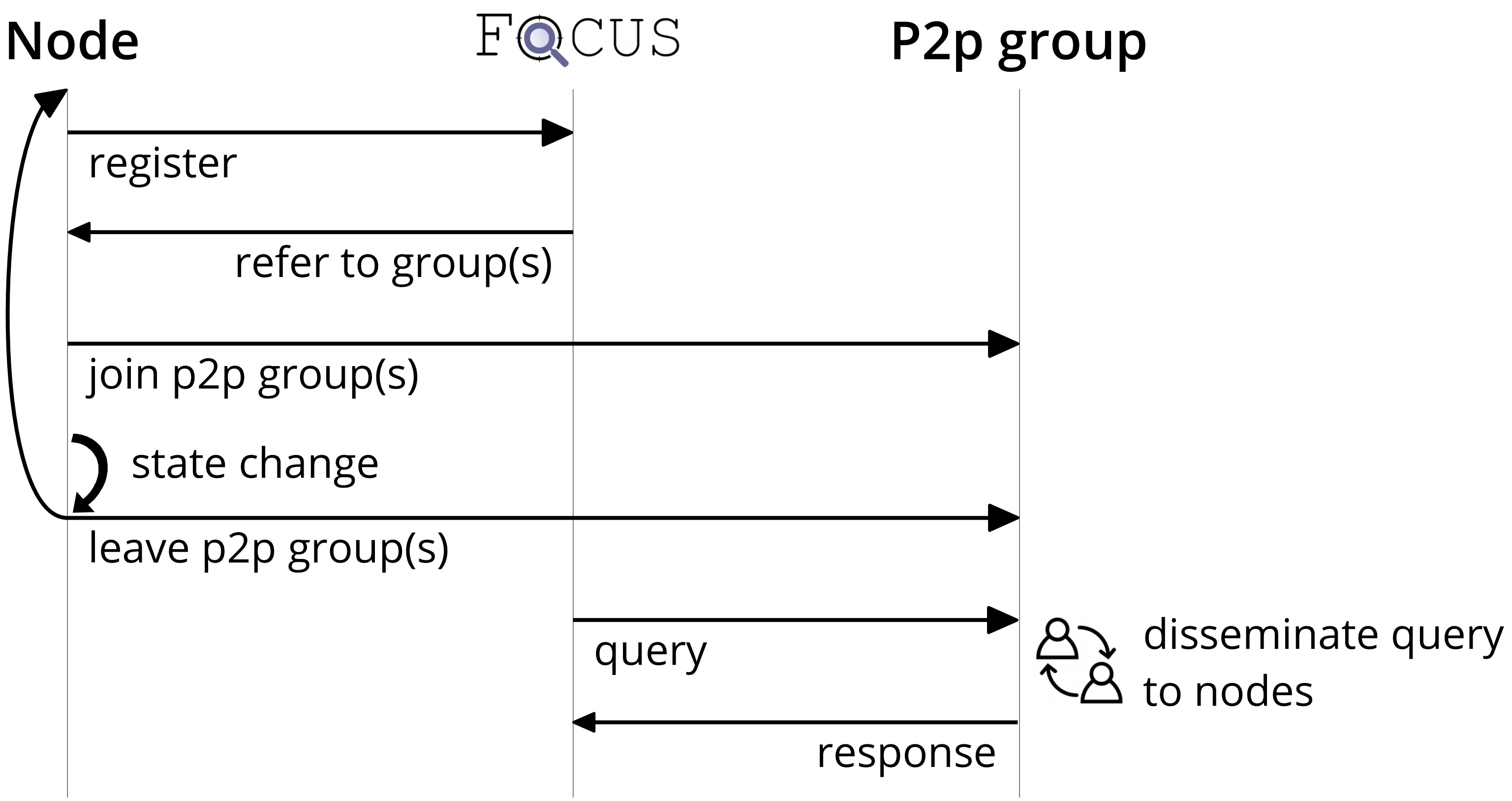
Dynamic Groups Management

Operations flow in FOCUS



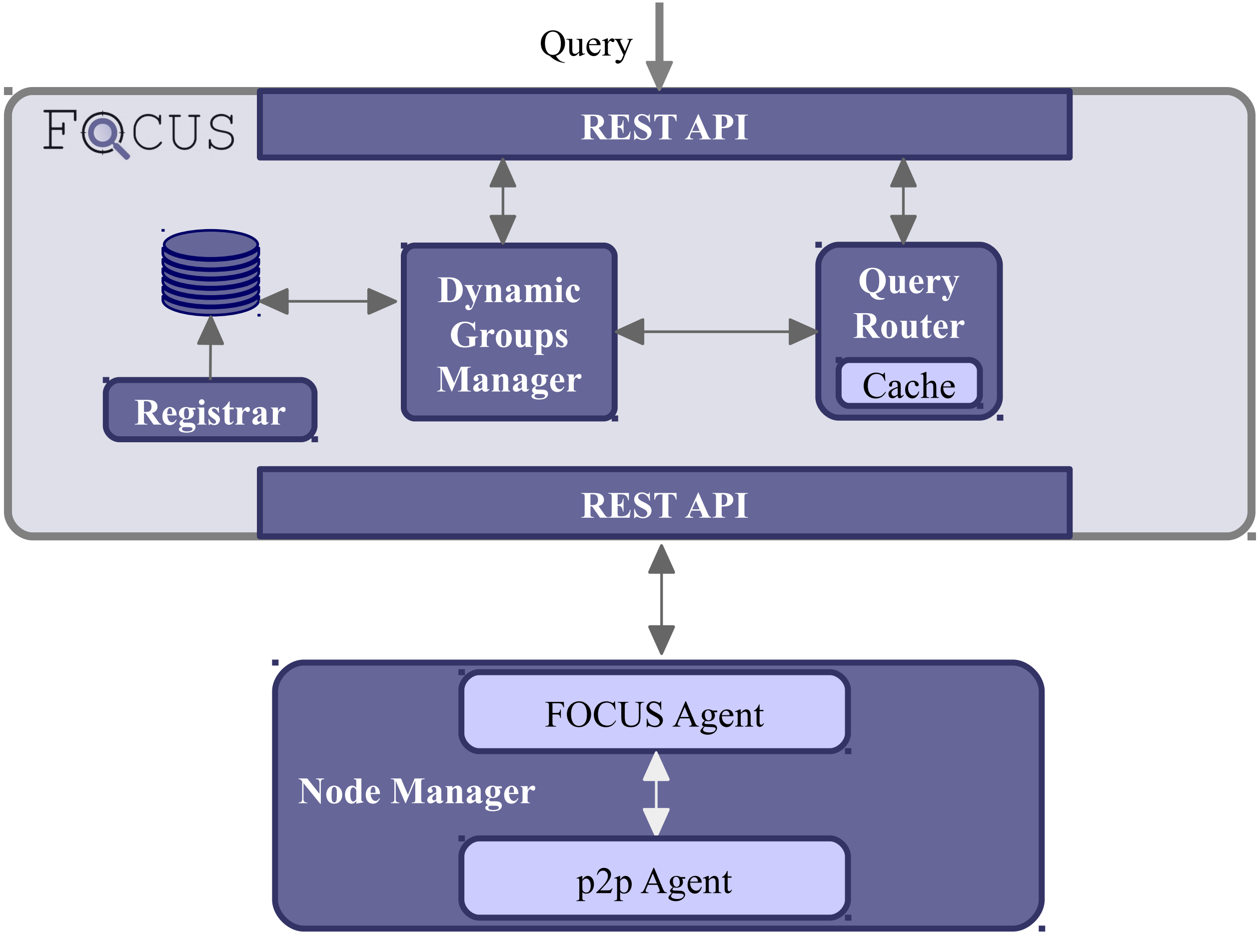
Dynamic Groups Management

Operations flow in FOCUS

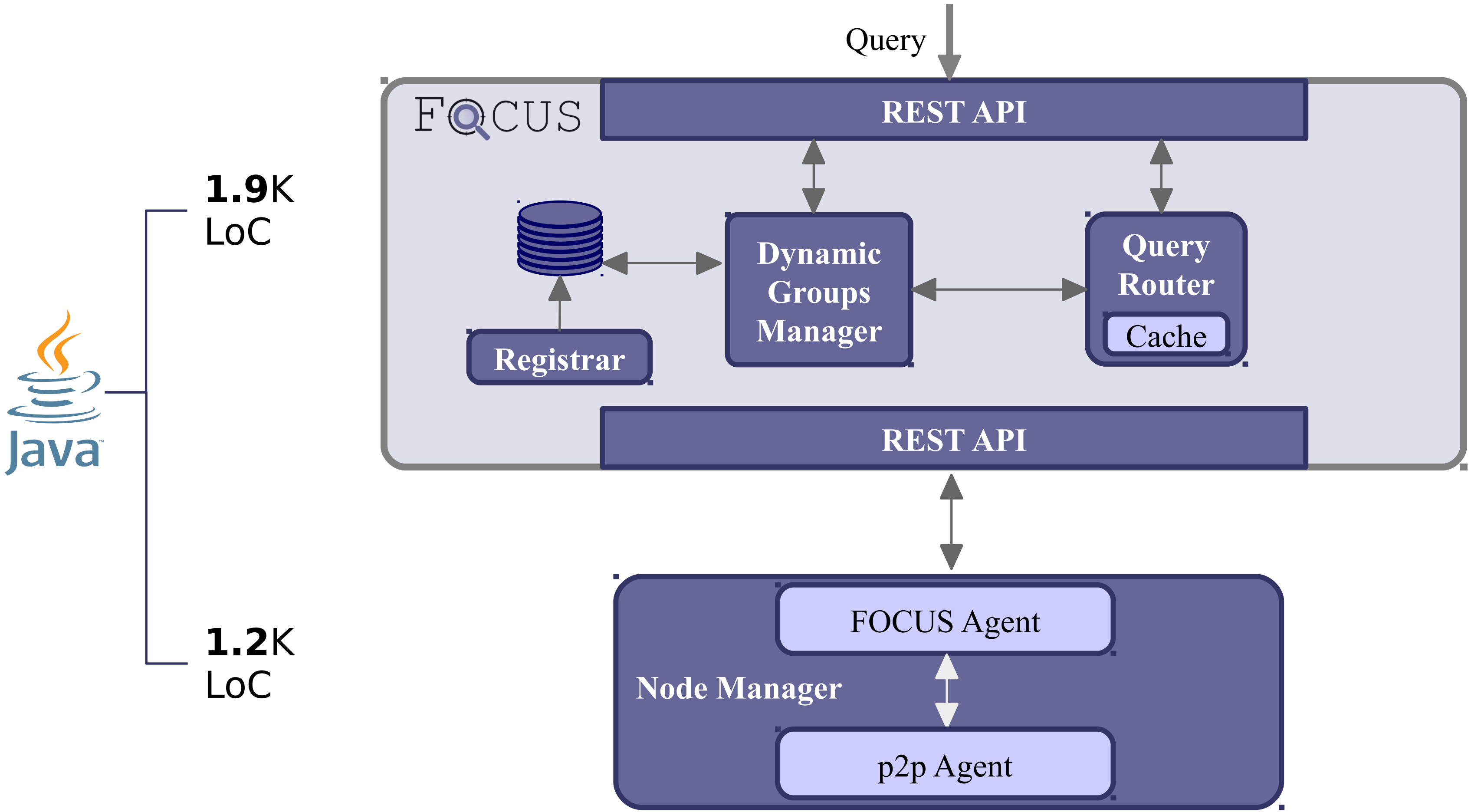


Implementation & Evaluation

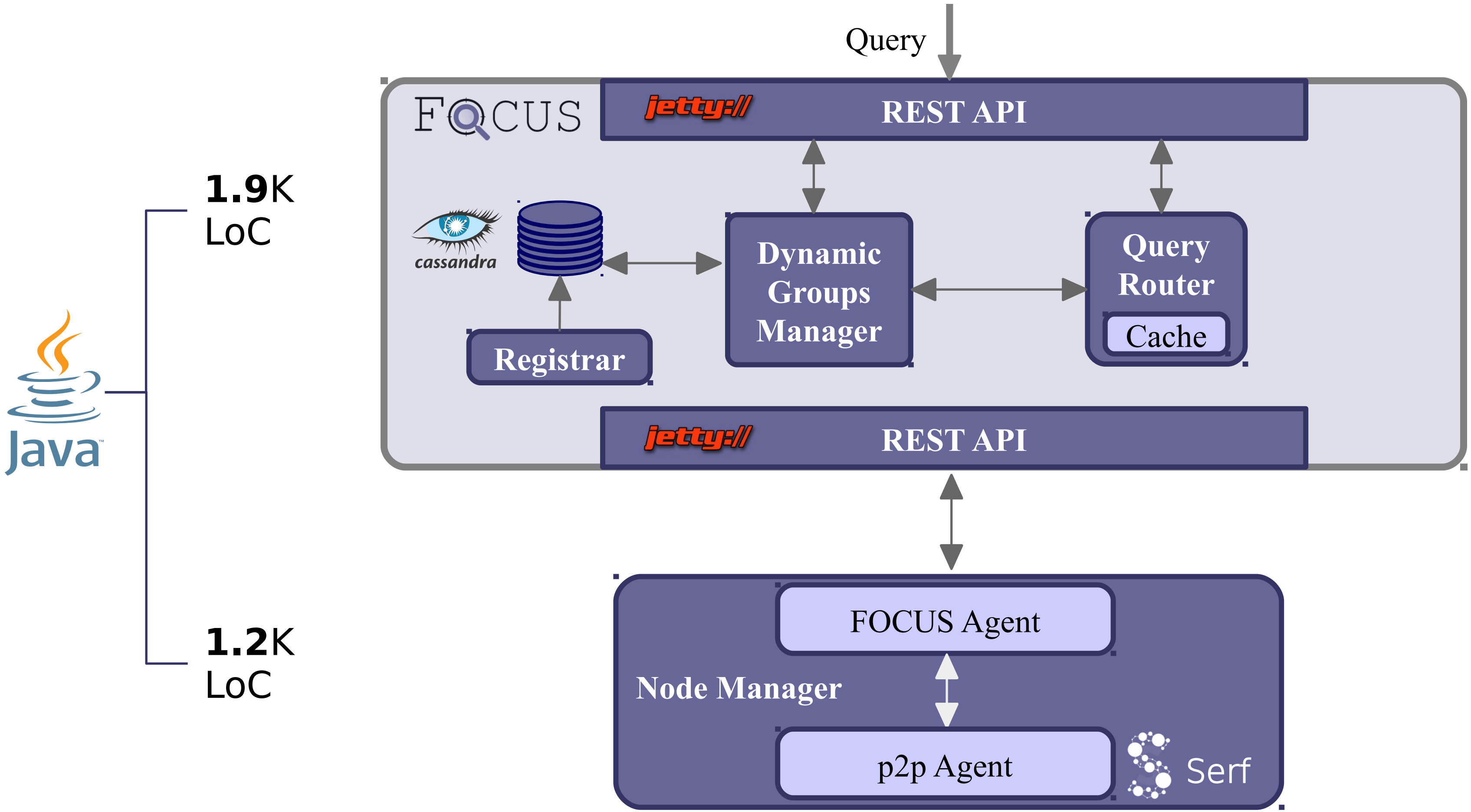
Implementation



Implementation



Implementation



Evaluation

- Deployed in Amazon EC2
- 4 regions: Canada, California, Ohio, Oregon
- In each region: 8 VMs (4 vCPUs, 16GB RAM)
- FOCUS server running in California (same VM config)
- Testing up to 1600 simulated node agents

FOCUS vs. Other Approaches

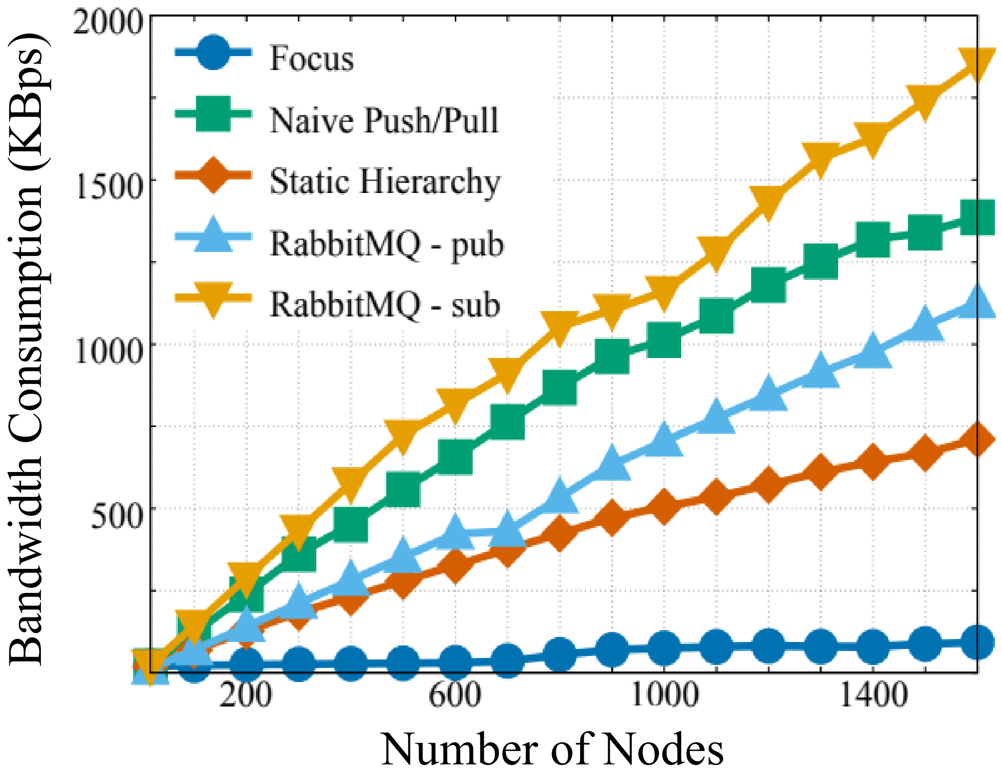
Measuring BW Consumption at the Query Server
(frequency = 1 query/update per second)

Naive Push/Pull

Static Hierarchy

RabbitMQ (Publish)

RabbitMQ (Subscribe)



FOCUS vs. Other Approaches

Measuring BW Consumption at the Query Server
(frequency = 1 query/update per second)

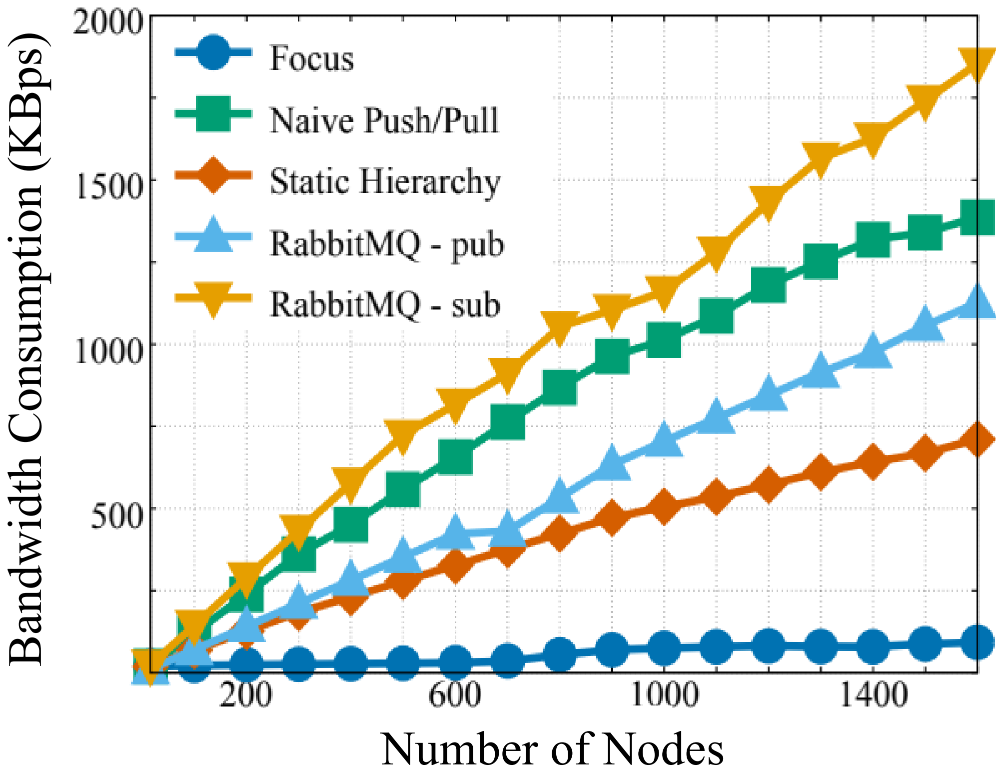
Naive Push/Pull

Static Hierarchy

RabbitMQ (Publish)

RabbitMQ (Subscribe)

Adding a layer of intermediate nodes acting as aggregators



FOCUS vs. Other Approaches

Measuring BW Consumption at the Query Server
(frequency = 1 query/update per second)

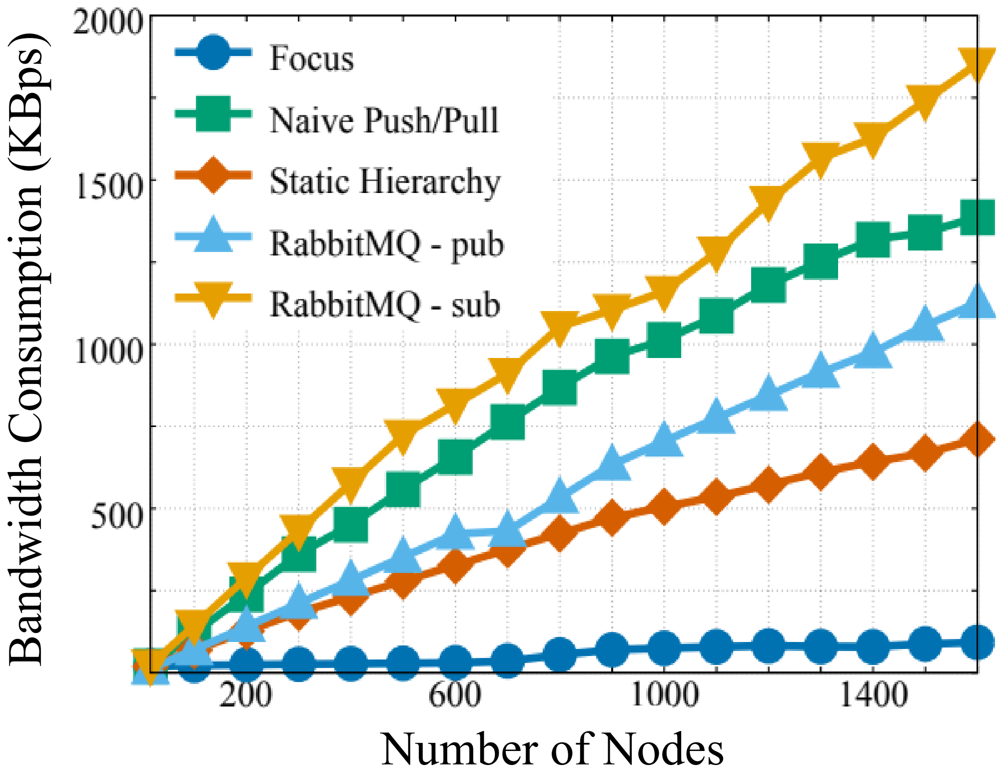
Naive Push/Pull

Static Hierarchy

RabbitMQ (Publish)

RabbitMQ (Subscribe)

Nodes publish their state (i.e.,fancy push)



FOCUS vs. Other Approaches

Measuring BW Consumption at the Query Server
(frequency = 1 query/update per second)

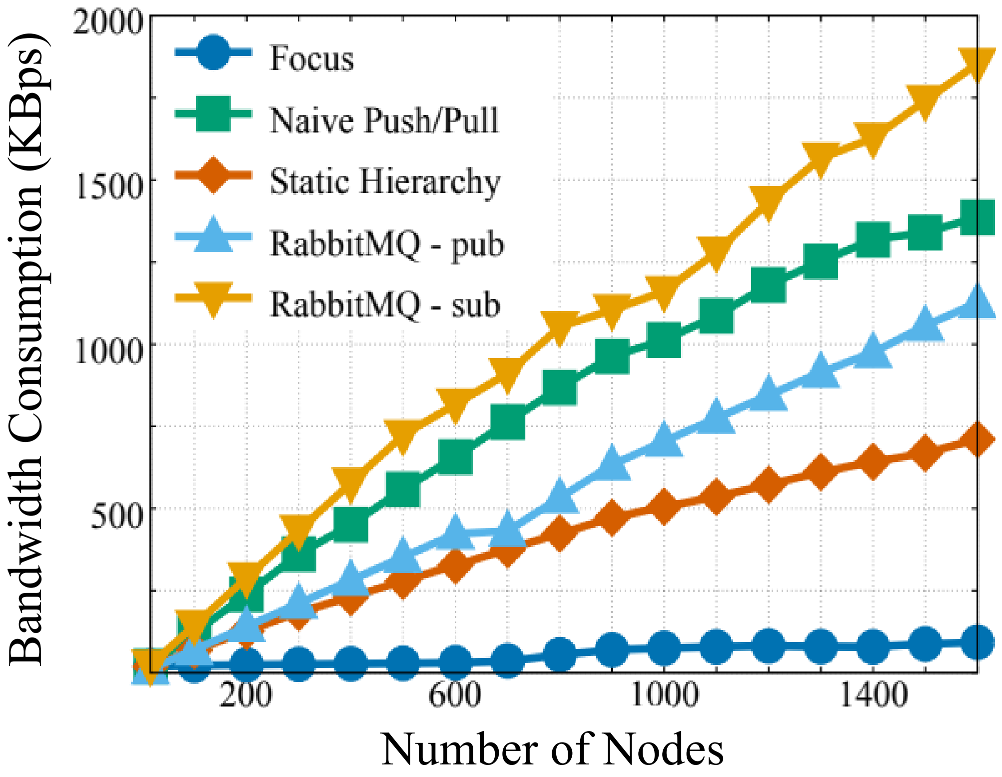
Naive Push/Pull

Static Hierarchy

RabbitMQ (Publish)

RabbitMQ (Subscribe)

Nodes subscribe for queries



FOCUS vs. Other Approaches

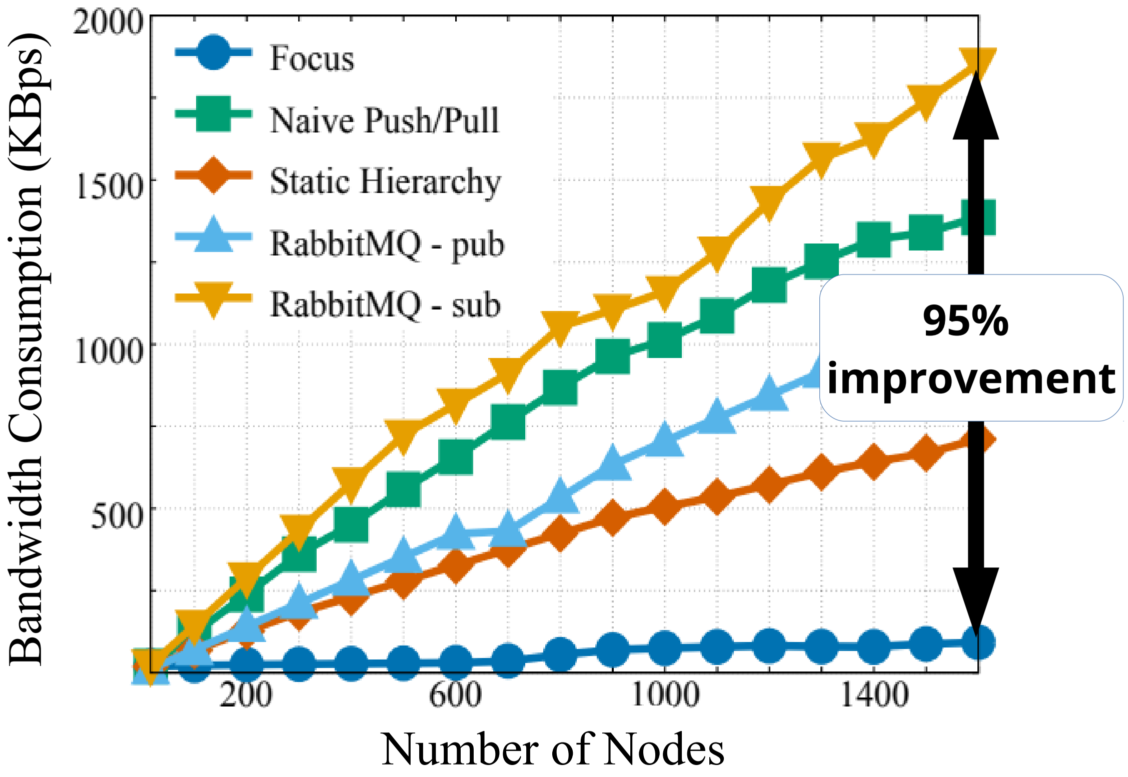
Measuring BW Consumption at the Query Server
(frequency = 1 query/update per second)

Naive Push/Pull

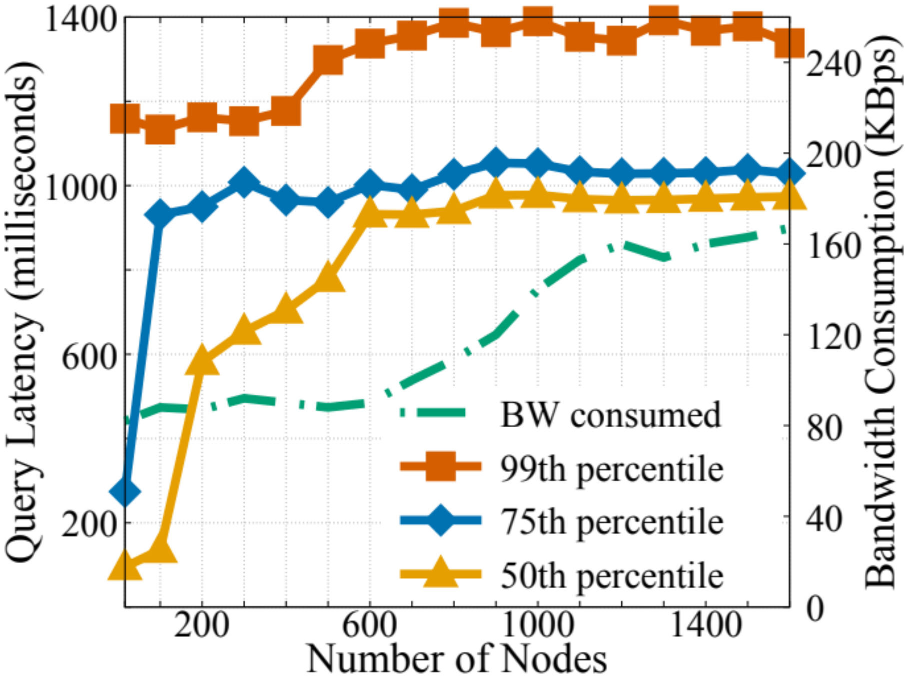
Static Hierarchy

RabbitMQ (Publish)

RabbitMQ (Subscribe)



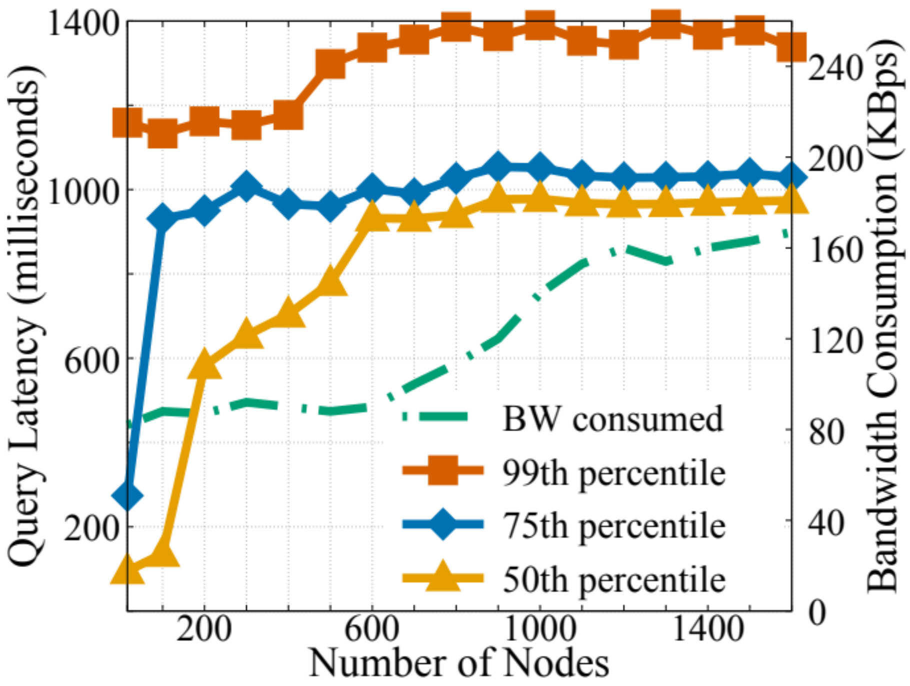
FOCUS with Real-world Cloud Traces*



* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

FOCUS with Real-world Cloud Traces*

75K OpenStack VM placement requests

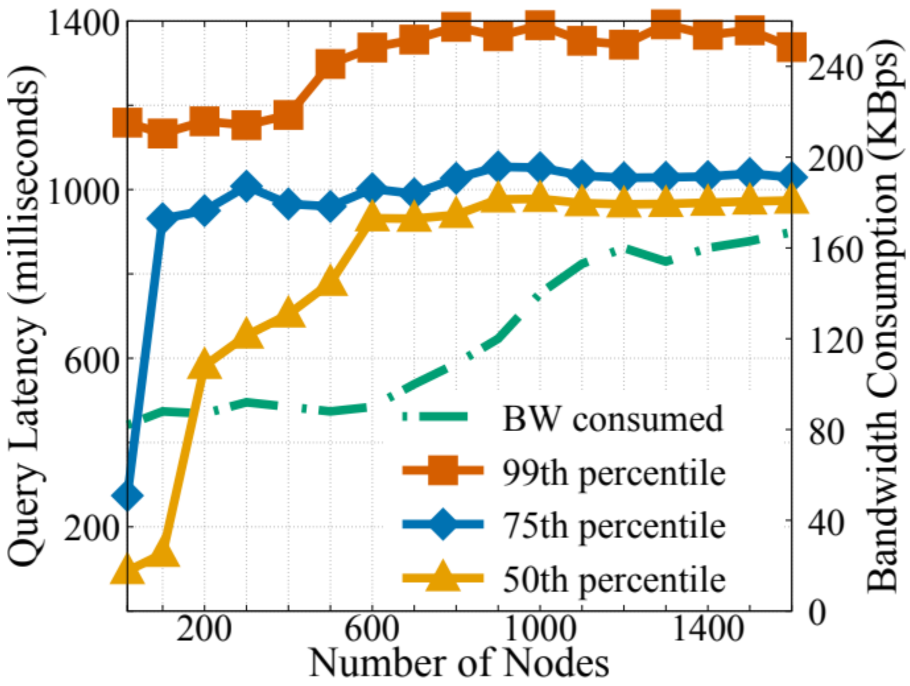


* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

FOCUS with Real-world Cloud Traces*

75K OpenStack VM placement requests

Replayed at accelerated rate (15,000x)



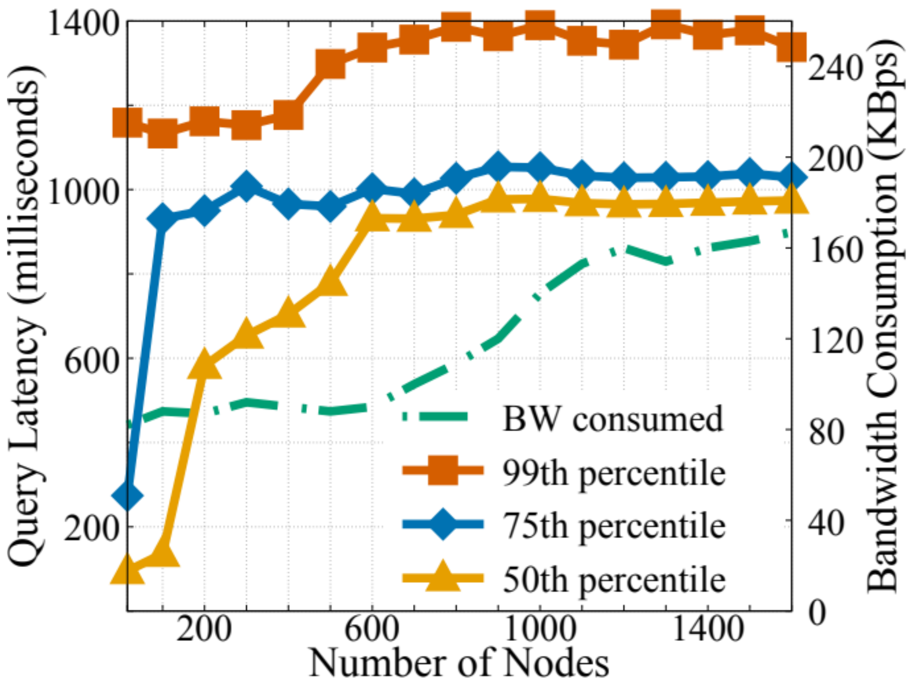
* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

FOCUS with Real-world Cloud Traces*

75K OpenStack VM placement requests

Replayed at accelerated rate (15,000x)

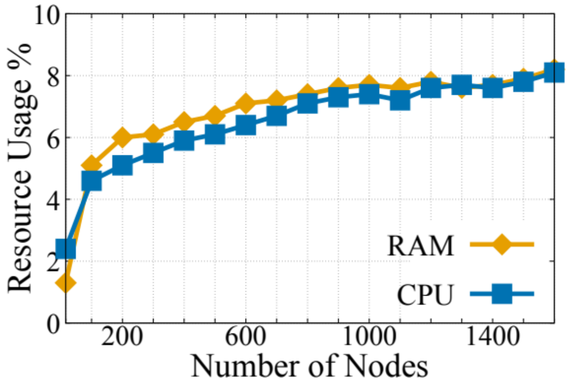
Latency stabilizes after 600 nodes
→ because group size is capped (~150 nodes per group)



* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

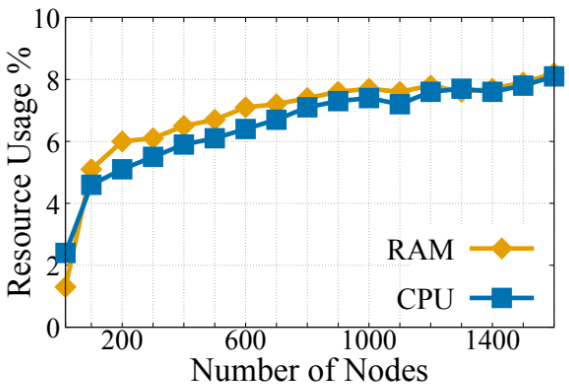
Microbenchmarks

Resource usage of the FOCUS server (40 queries/s)

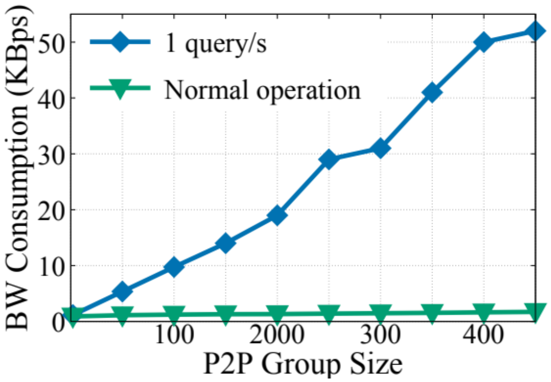


Microbenchmarks

Resource usage of the FOCUS server (40 queries/s)

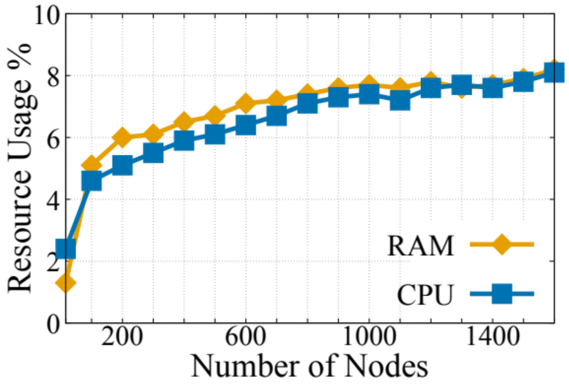


Overhead imposed by node agent (KBps)

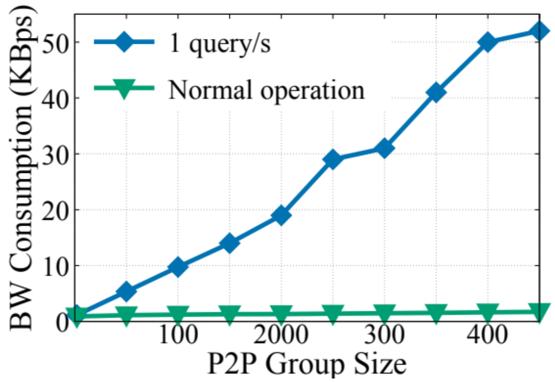


Microbenchmarks

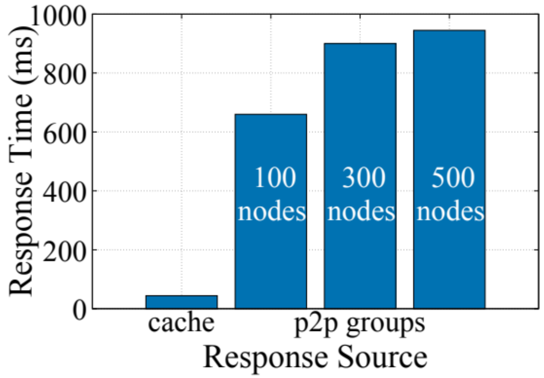
Resource usage of the FOCUS server (40 queries/s)



Overhead imposed by node agent (KBps)



Query response time for different group sizes



Conclusion

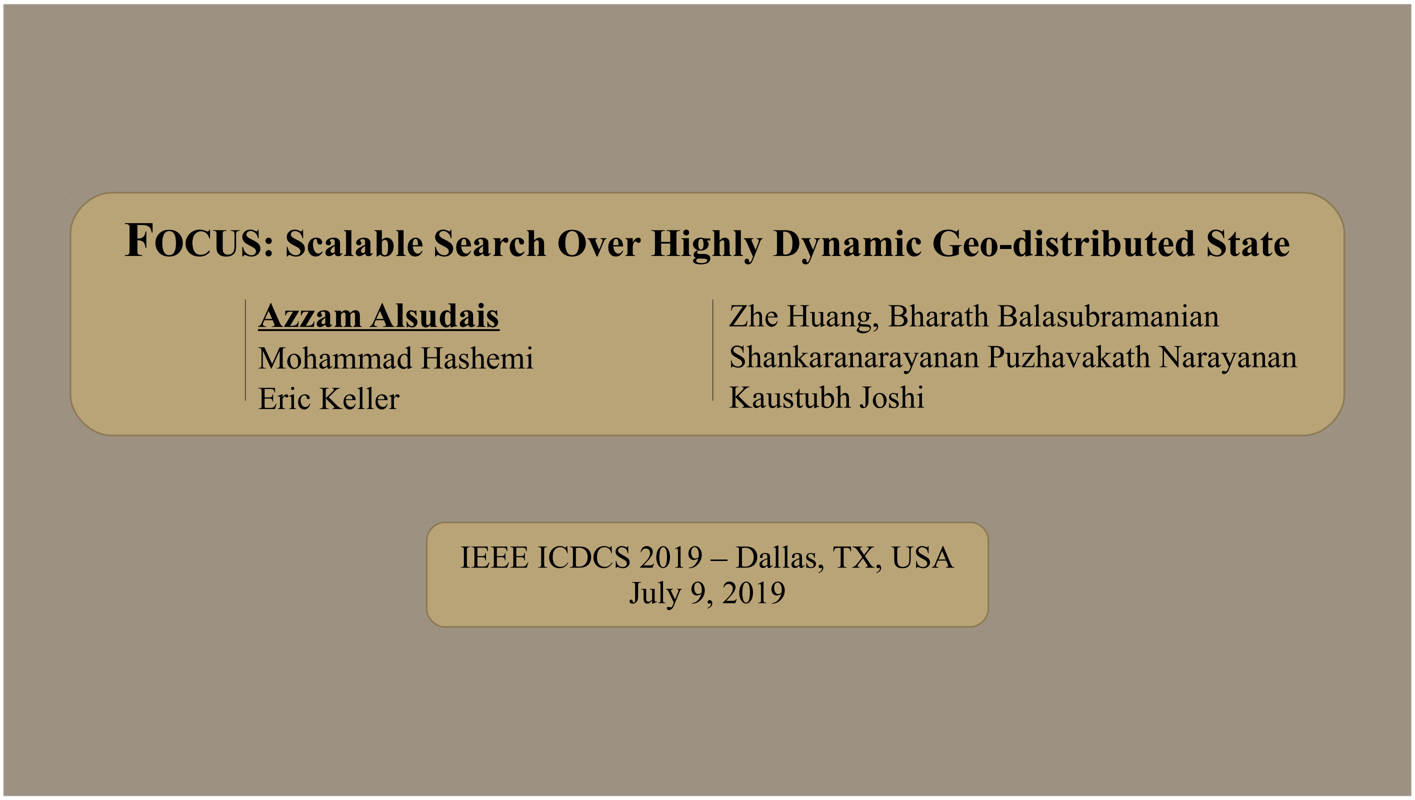
- **Current systems’ scalability is limited**
  - This is due to tightly-coupled node management

Conclusion

- **Current systems' scalability is limited**
  - This is due to tightly-coupled node management
- **FOCUS is scalable search service**
  - Employs a *loosely-coupled* node management (p2p)
  - *Scales* better than current approaches (15x improvement)
  - Imposes *minimal* overhead on nodes
  - *Integrates* well with current systems

Thank You!

Questions?



Today, I'm presenting FOCUS: a generic and scalable search service for distributed systems in general with a focus on cloud systems.

This work is done in collaboration between University of Colorado Boulder and AT&T Labs Research.

Why do systems need to find nodes?

First, let’s take a look at why systems need a search service to find a selection of their nodes that satisfy certain criteria.

Use Cases

Cloud Management

I’ll present 2 use-cases that motivate the need for a search service.

.

In cloud management platforms, we need to find nodes for various reasons.

Use Cases

Cloud Management

VM Provisioning

For instance, we need to find the best nodes to do things like VM placement and provisioning.

Use Cases

Cloud Management

VM Provisioning

VM Migration

Also, an extension to the previous example is that we need to find nodes to do VM migration.

For example, we need to find nodes that have enough capacities to migrate live Vms.

Use Cases

Cloud Management

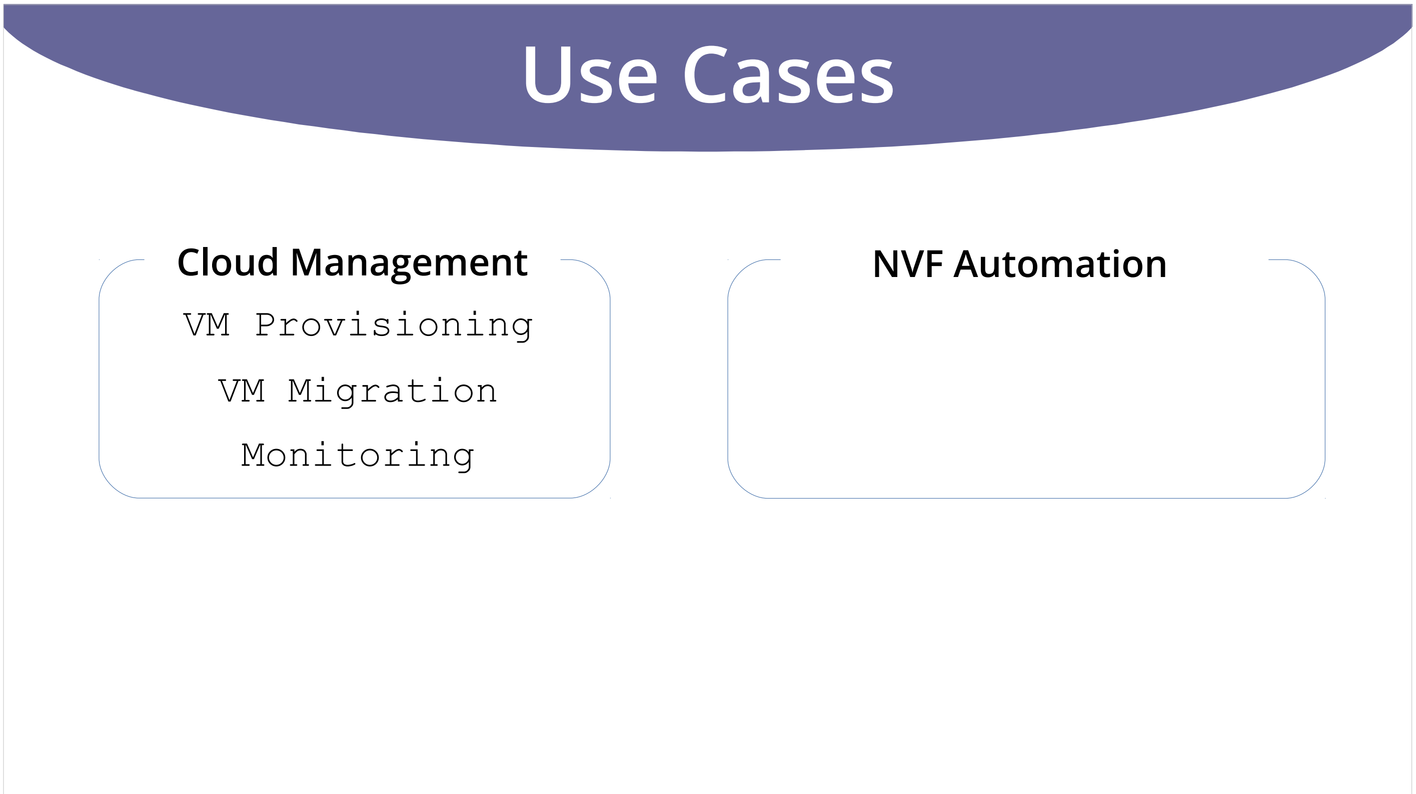
VM Provisioning

VM Migration

Monitoring

Doing Health-checks and Monitoring, in general, is critical in Datacenter environments.

For instance, we might want an alarm that gets triggered whenever there’s some overload on certain nodes so that we can do things like VM migration.



Another use-case for a search service is doing automation for Network Function Virtualization (or NFV) systems.

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

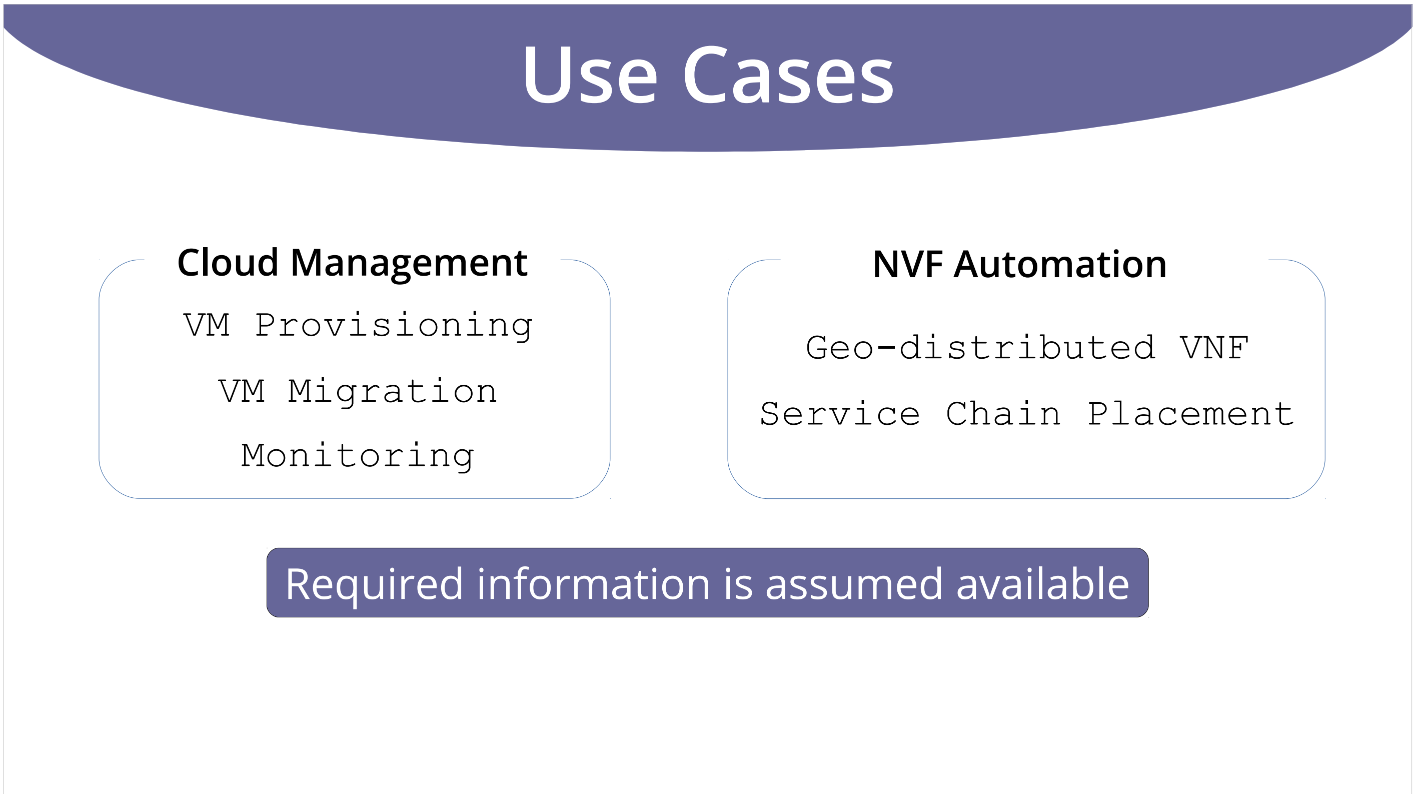
Geo-distributed VNF

Service Chain Placement

This includes performing VNF placement across geo-distributed sites.

To some extent, it’s similar to doing VM placement, but at a larger scale and with more complex requirements and policies.

In this use-case, we need to find nodes that are best suited to host our Network Functions.



So, in such use-cases, systems make the assumption that required information (like current capacities of the nodes) is always available whenever they wish.

Use Cases

Cloud Management

VM Provisioning

VM Migration

Monitoring

NVF Automation

Geo-distributed VNF

Service Chain Placement

Required information is assumed available

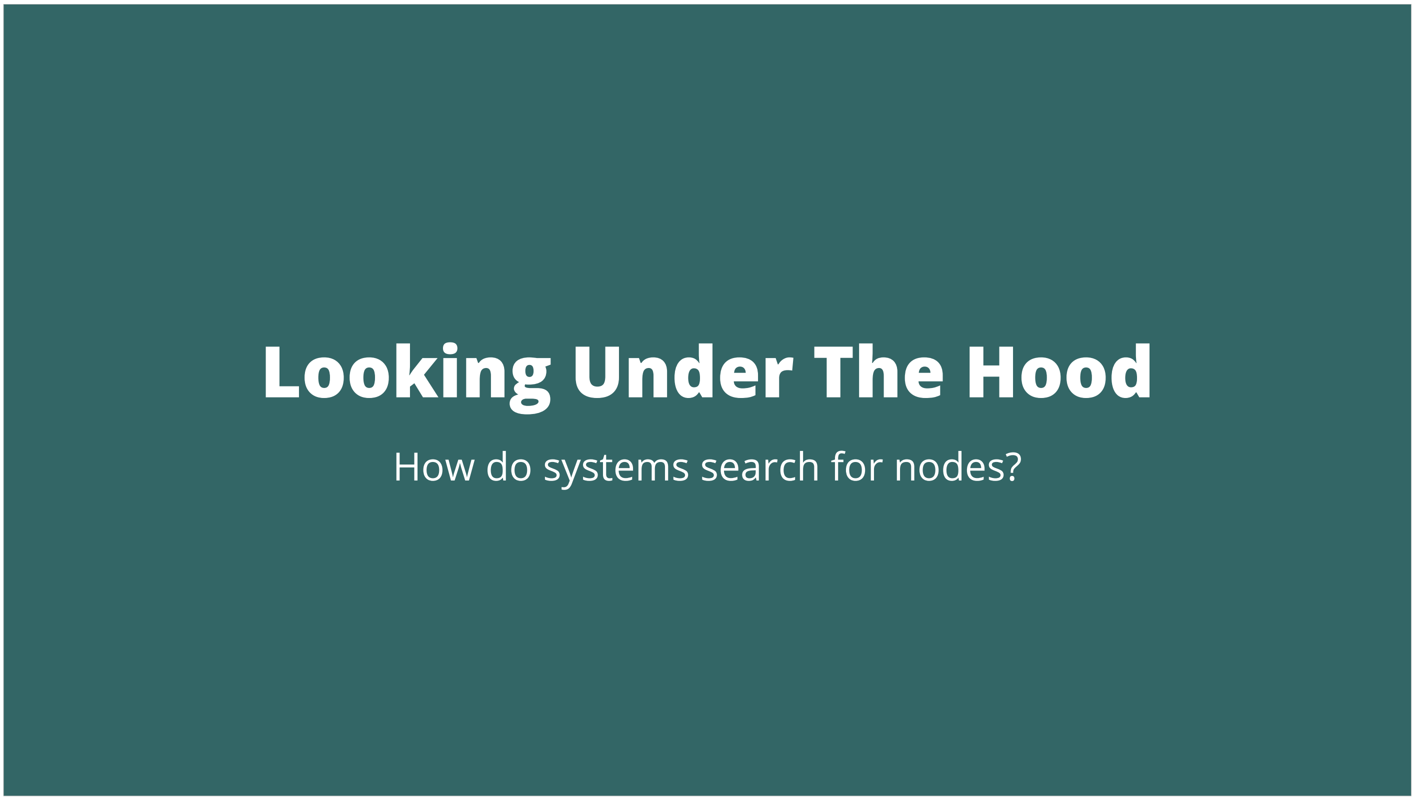
But **HOW** is node information collected?

Now, let’s take a look at how such systems collect this critical information in order to perform tasks such as VM, or VNF placement.

Outline

- How do systems find nodes?
- Limitations of current approaches
- FOCUS design
- Evaluation
- Conclusion

In the rest of the presentation, I’ll talk about:
- how systems find nodes
.
- also, we will look at why those are limited, especially when deploying them at a large scale.
.
- Then, I will present our solution, FOCUS, which leverages concepts from p2p systems to scale.
.
- At the end, I will conclude with our evaluation.



Ok, so let’s go back to the question:
How do systems find nodes that satisfy certain criteria?

Node finding in

And to be practical, I’ll use OpenStack to better demonstrate the answer.

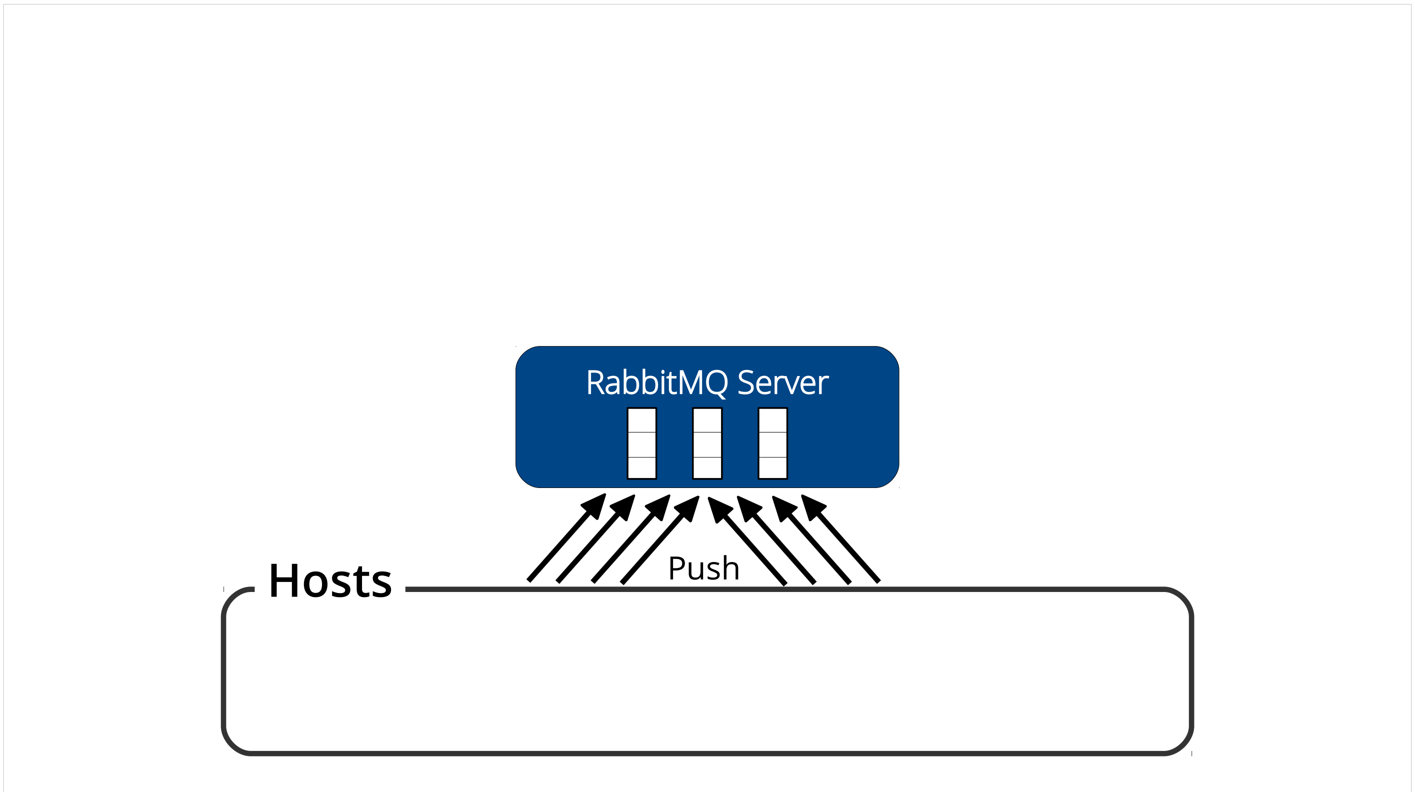
.

For those who are not familiar with OpenStack, it is considered the de-facto standard for managing cloud infrastructures.

In OpenStack, there are physical hosts that comprise the system.

.

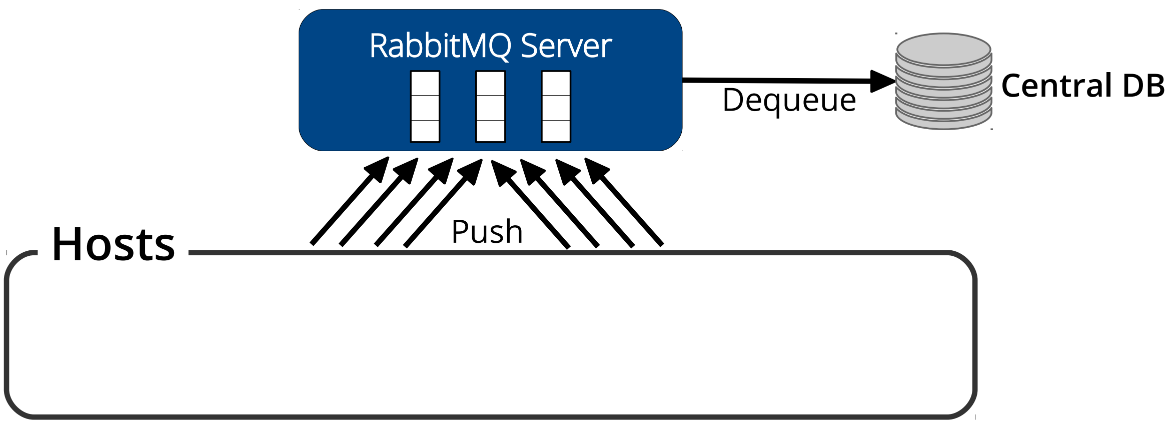
They are called Nova compute nodes, which are used as resources for things like VM placement.



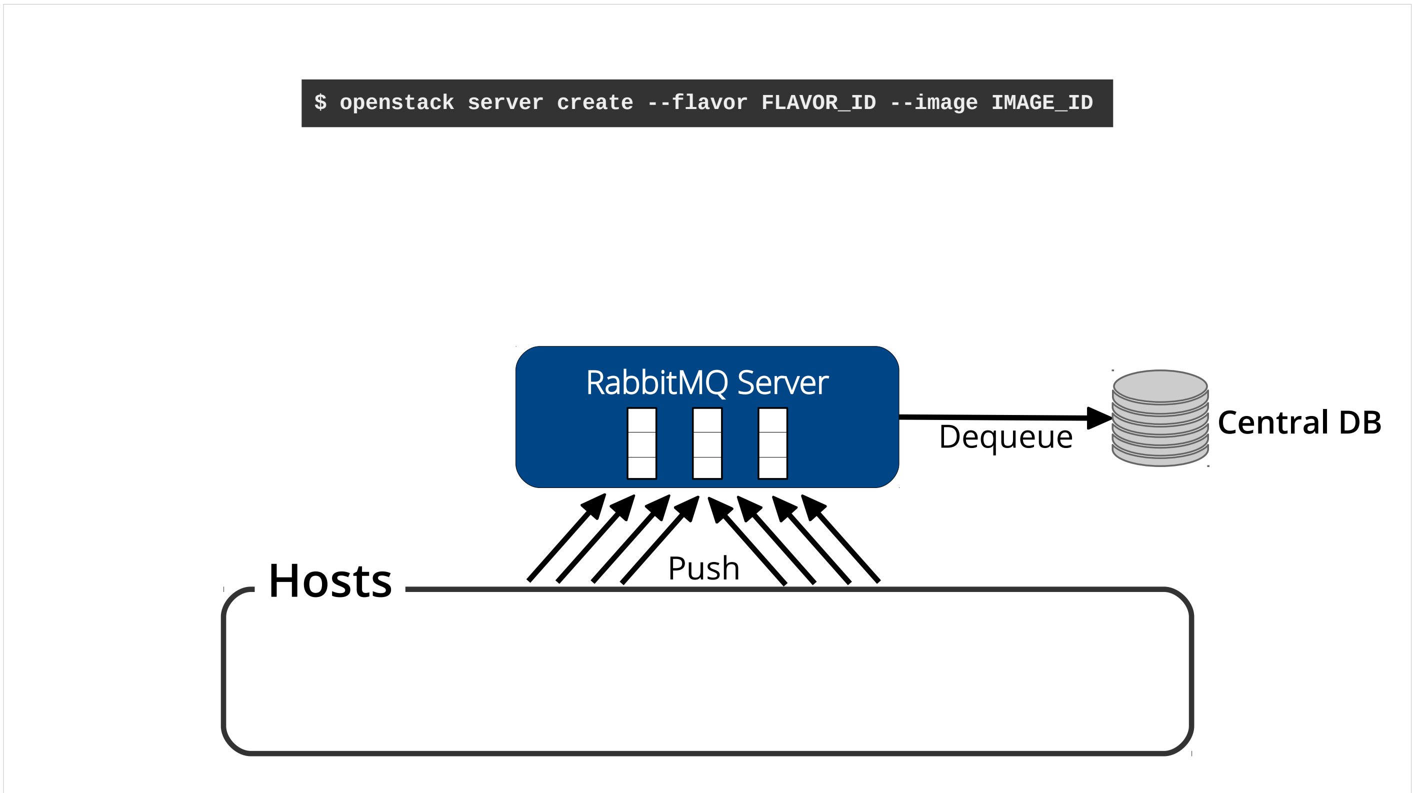
To obtain a current view of the system so that we know which nodes have which capacities, nodes periodically push their status to a special node that serves as a message bus.

.

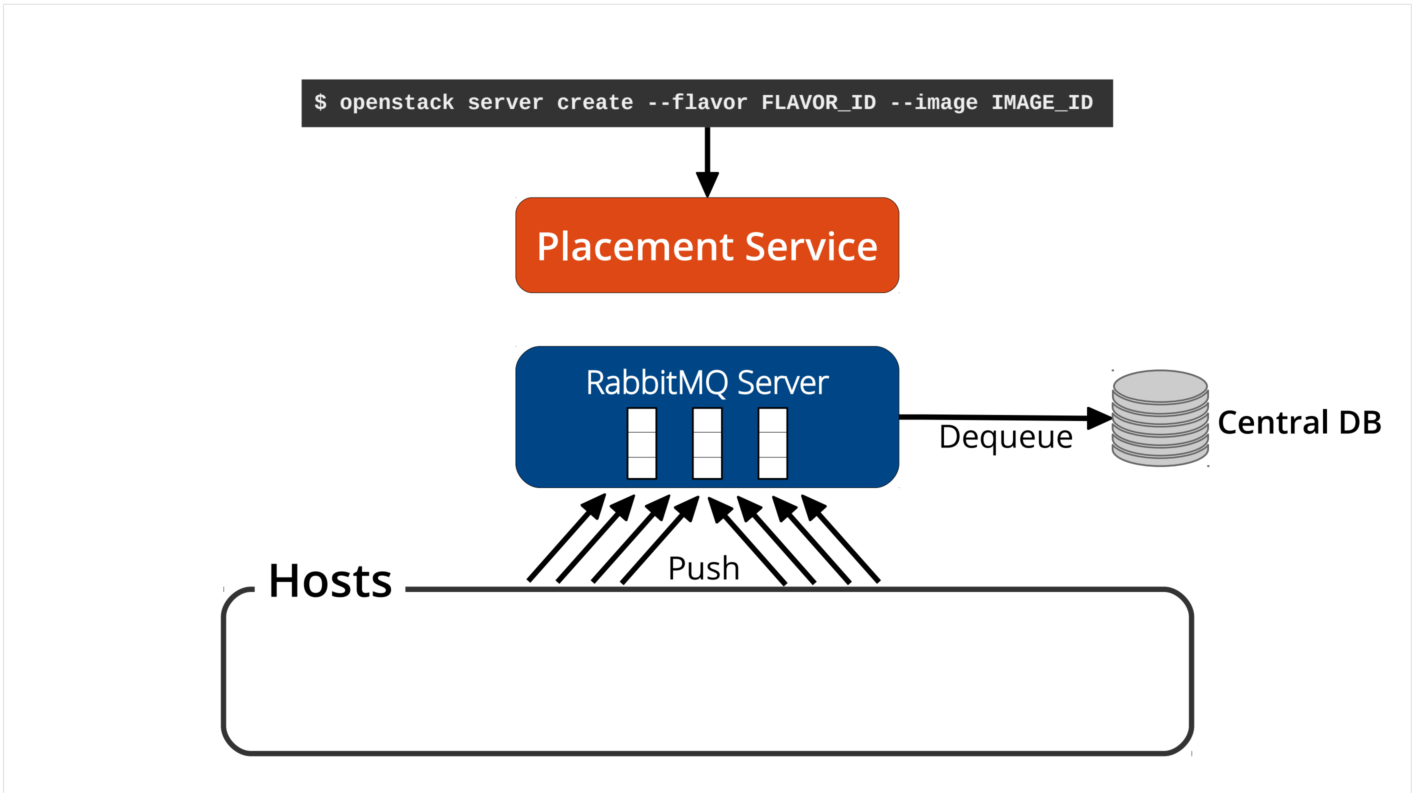
In OpenStack, RabbitMQ (a pub-sub service) is used by default.



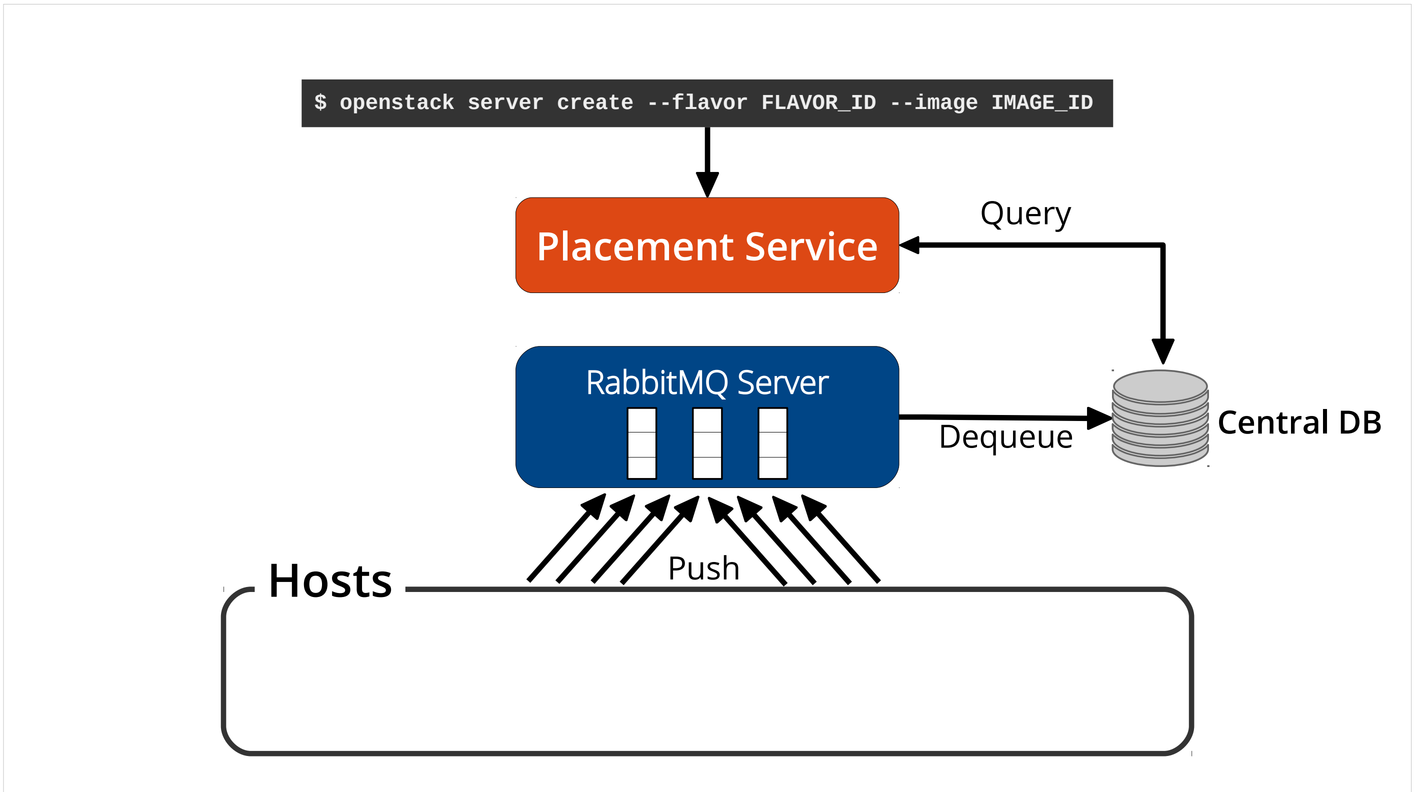
Then, a worker will dequeue information from the RabbitMQ node, and store it on a central database.



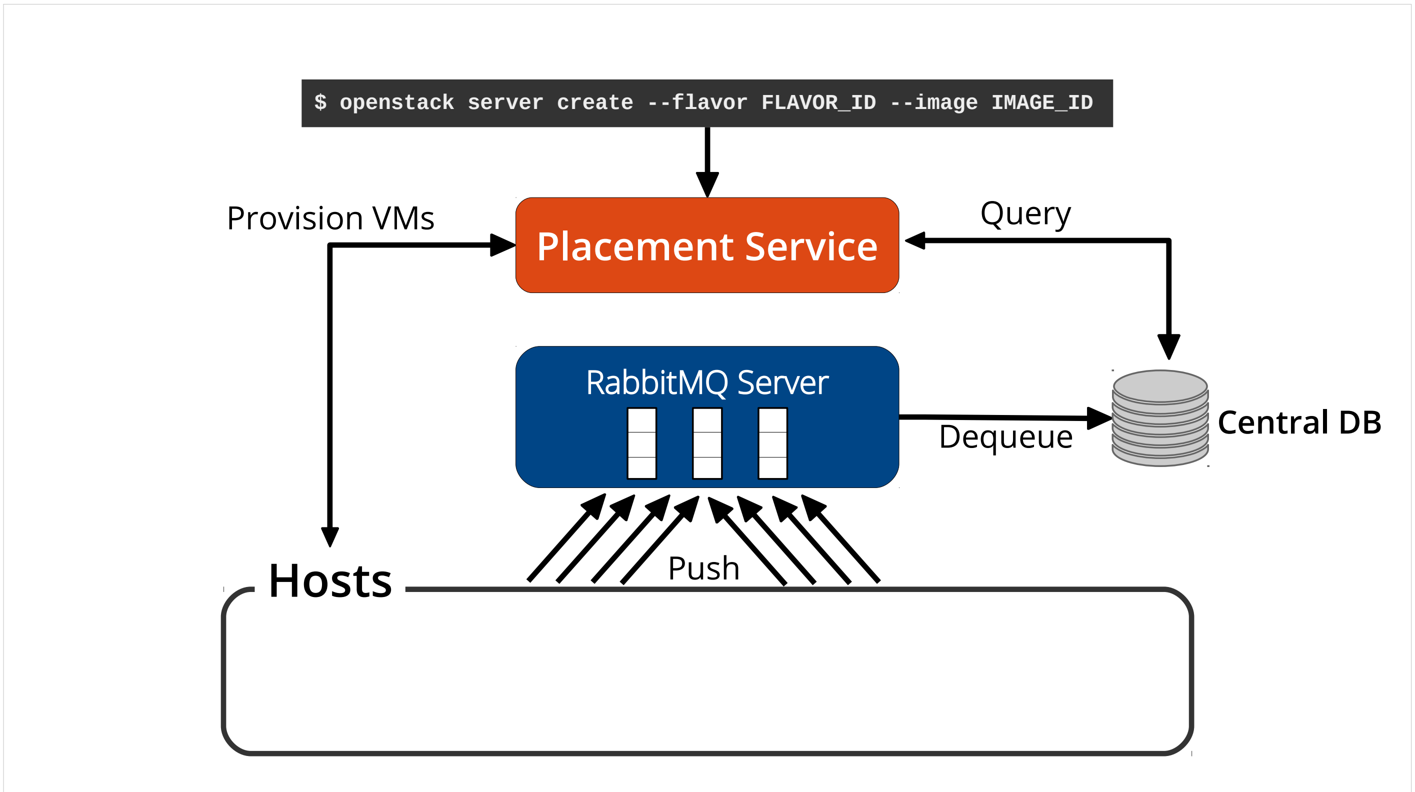
Now, when we issue a command like this to
OpenStack (where we want to provision some VM)
.
CLICK



It will go through OpenStack's placement service,
which parses the request,
And then
.
CLICK



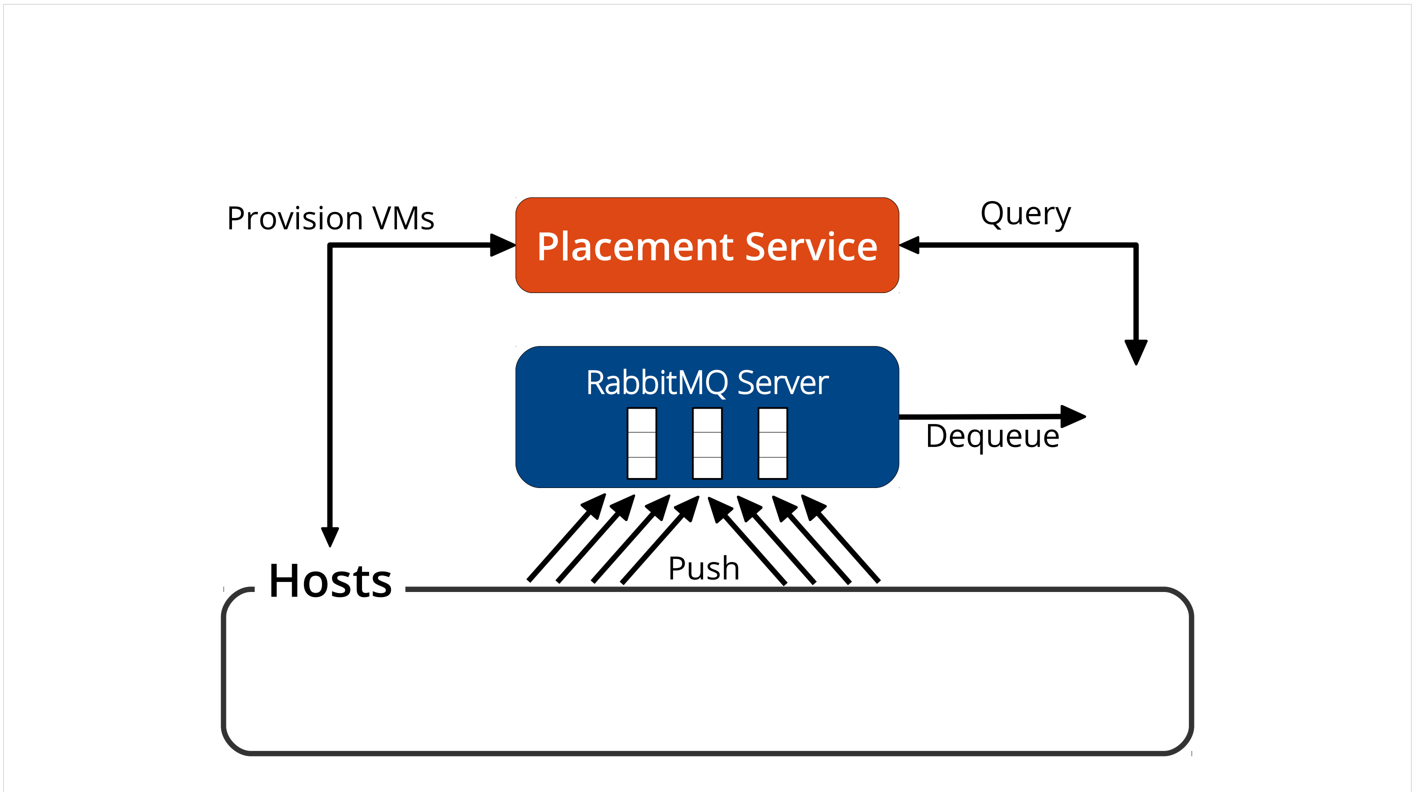
It will query the central Database to get host resource information.



It uses that information, then, to provision the VM using any placement logic (e.g., first-fit, greedy, etc).

Limitations of Current Approaches

Everything might seem to be fine. However, there are fundamental limitations to such approaches.



Specifically, there are 2 problems with this design.

.

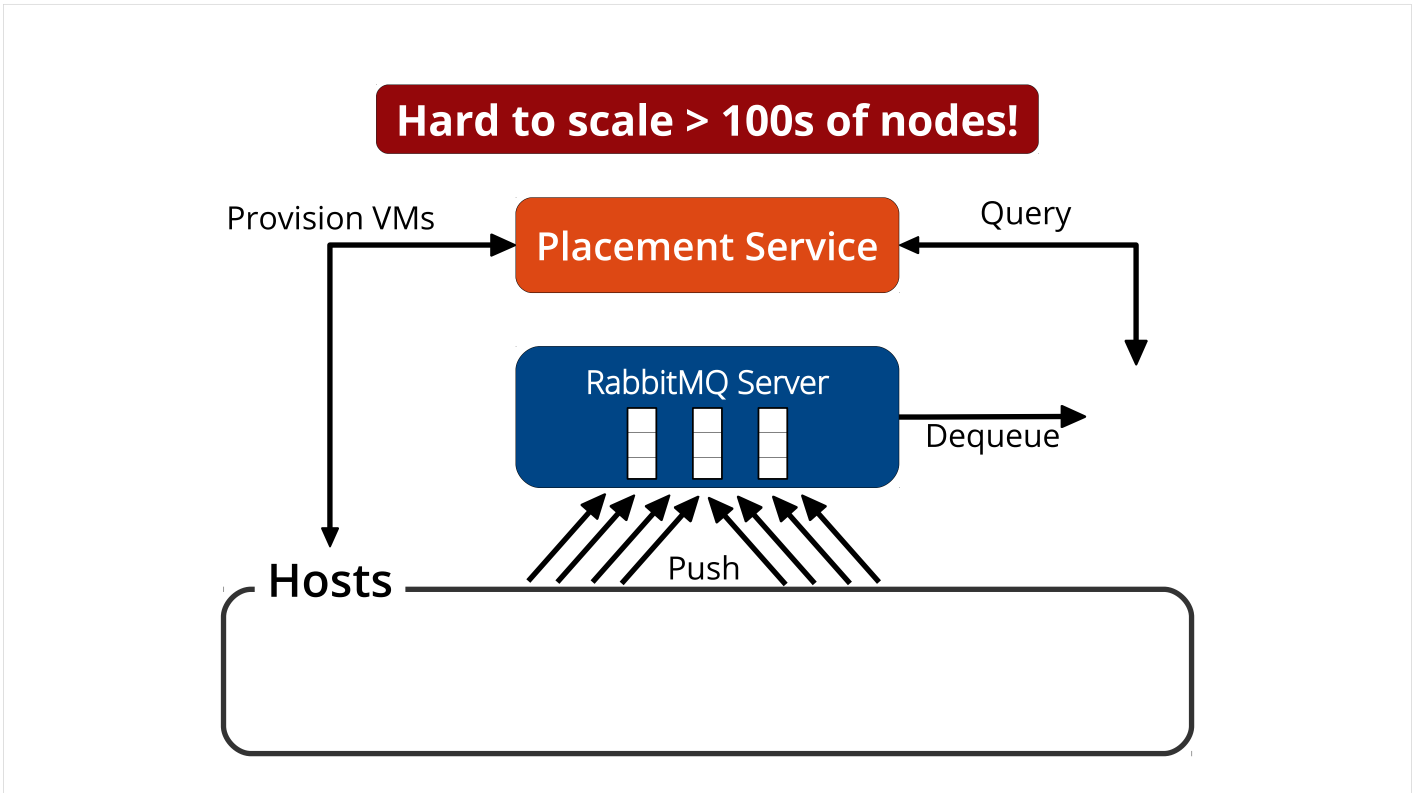
First: The information we get from the local DB can be stale. And thus, it might cause incorrect behavior when we act on outdated information.

.

Another, and more critical, problem that we have identified is that …

.

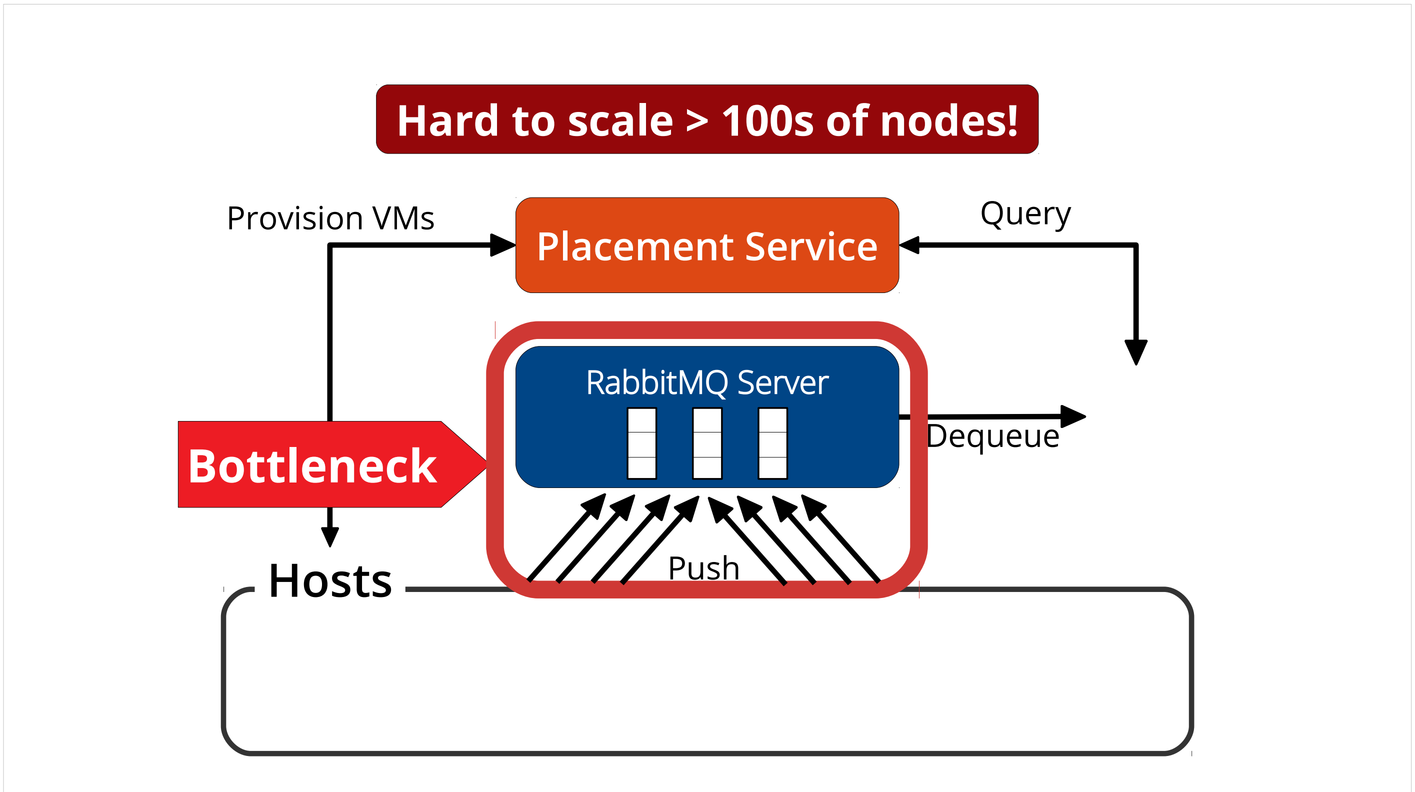
CLICK



It’s hard to scale such deployment beyond a few hundreds of hosts.

.

This is clearly a small number when we talk about multi-site clouds and geo-distributed network services.



Through some experimentation, we conclude that the bottleneck here is the use of message buses such as RabbitMQ.

.

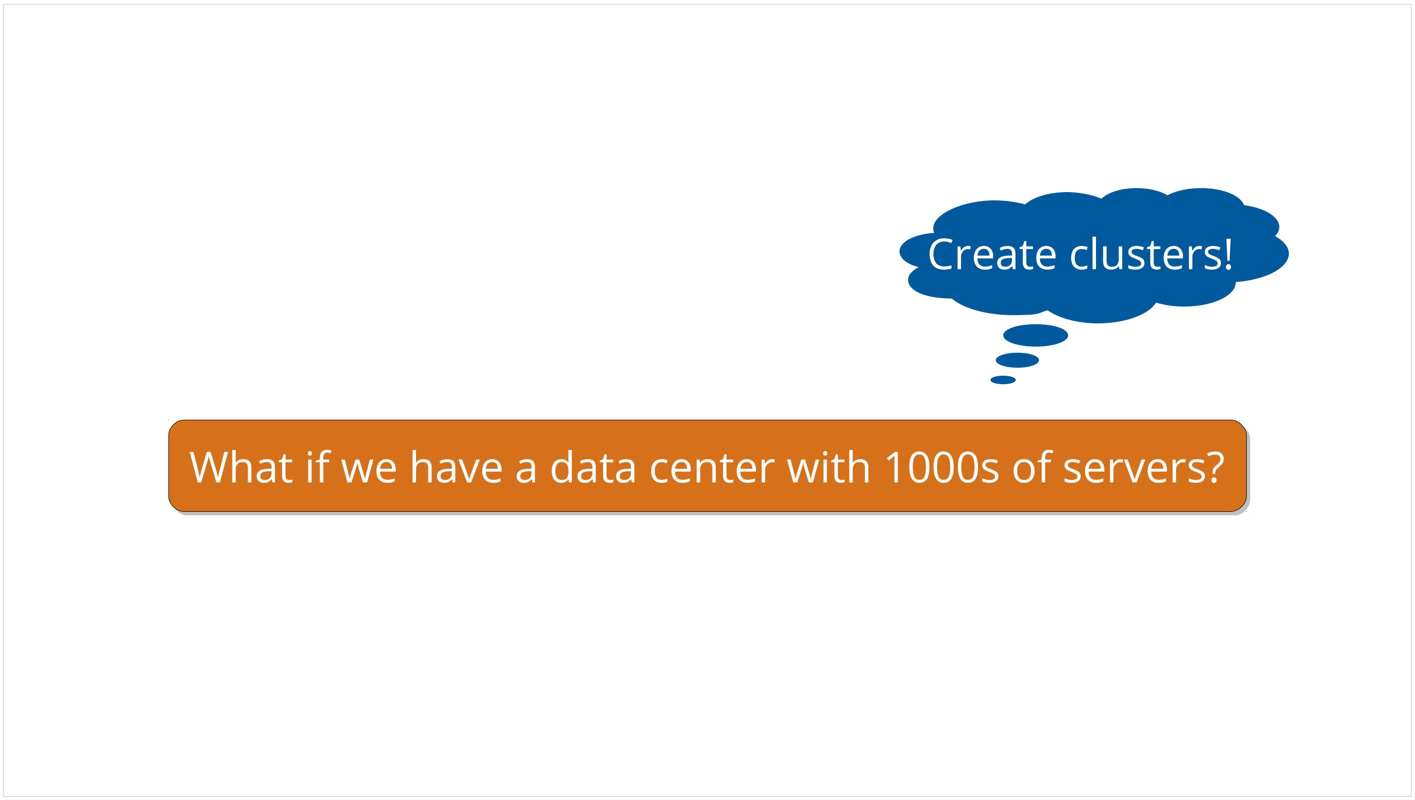
Our experiments are also backed by experiments performed by the OpenStack community.

.

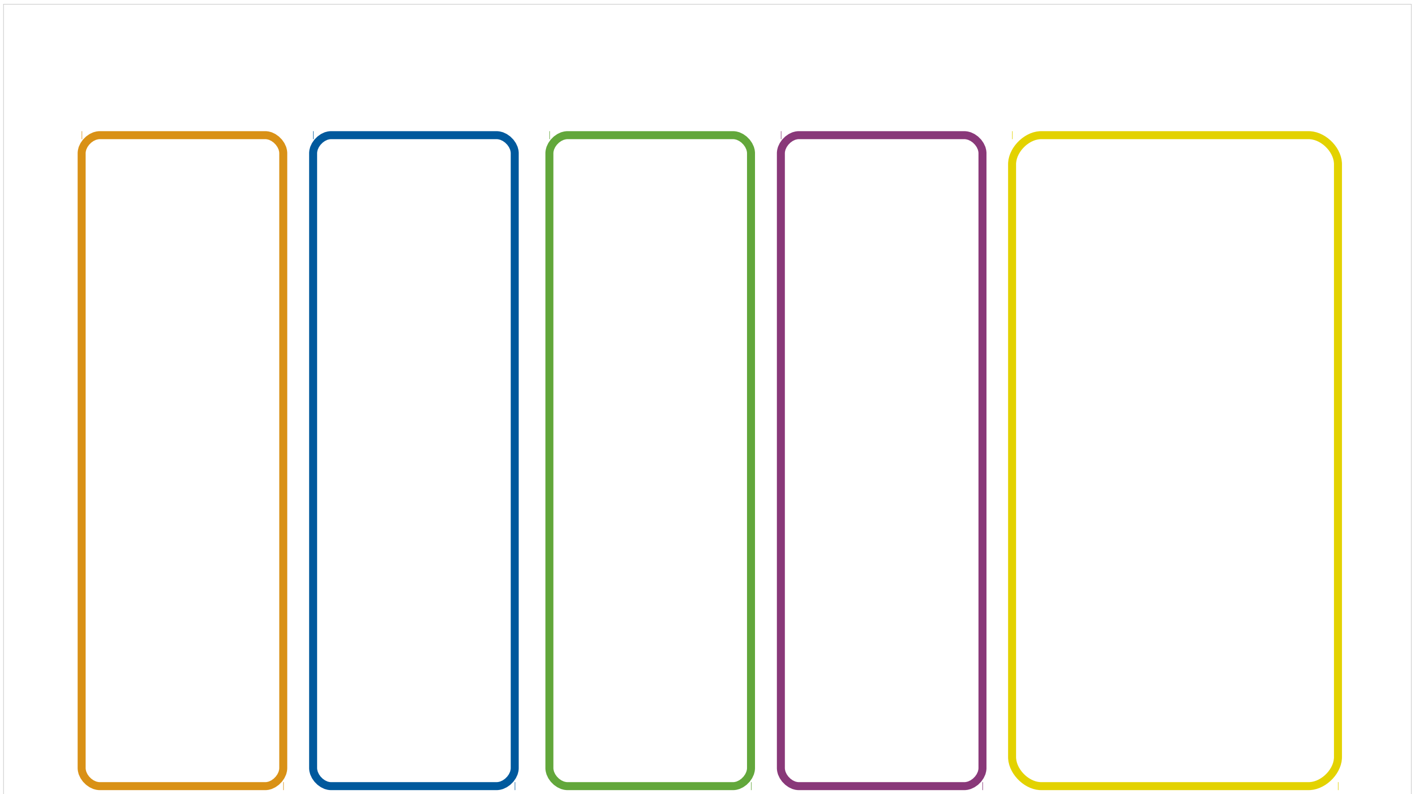
You can find the details of our experiment in the motivation section of the paper.

What if we have a data center with 1000s of servers?

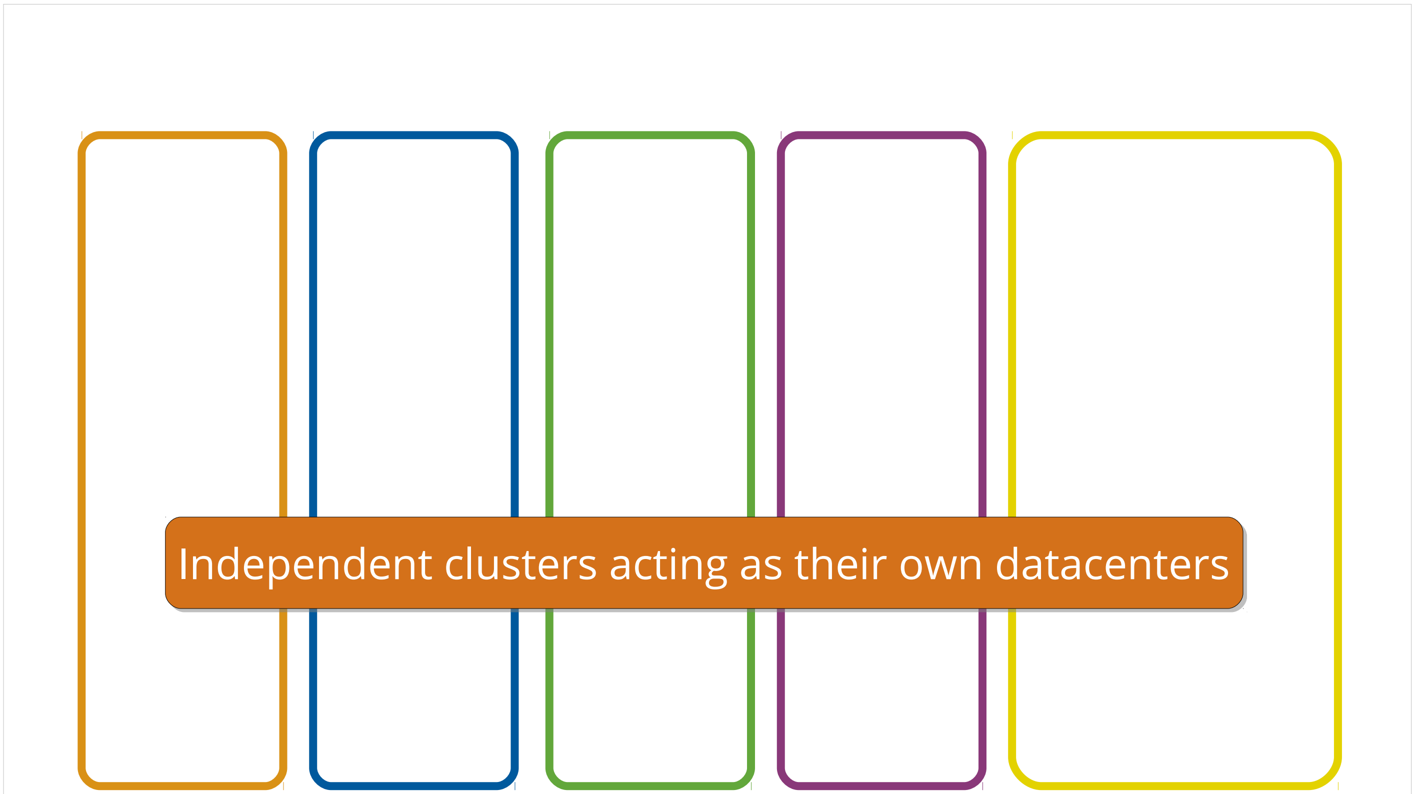
One might wonder, how can we support datacenters with thousands of nodes?



A typical solution would be to create clusters inside the datacenter



So, here we divide the datacenter into multiple clusters.

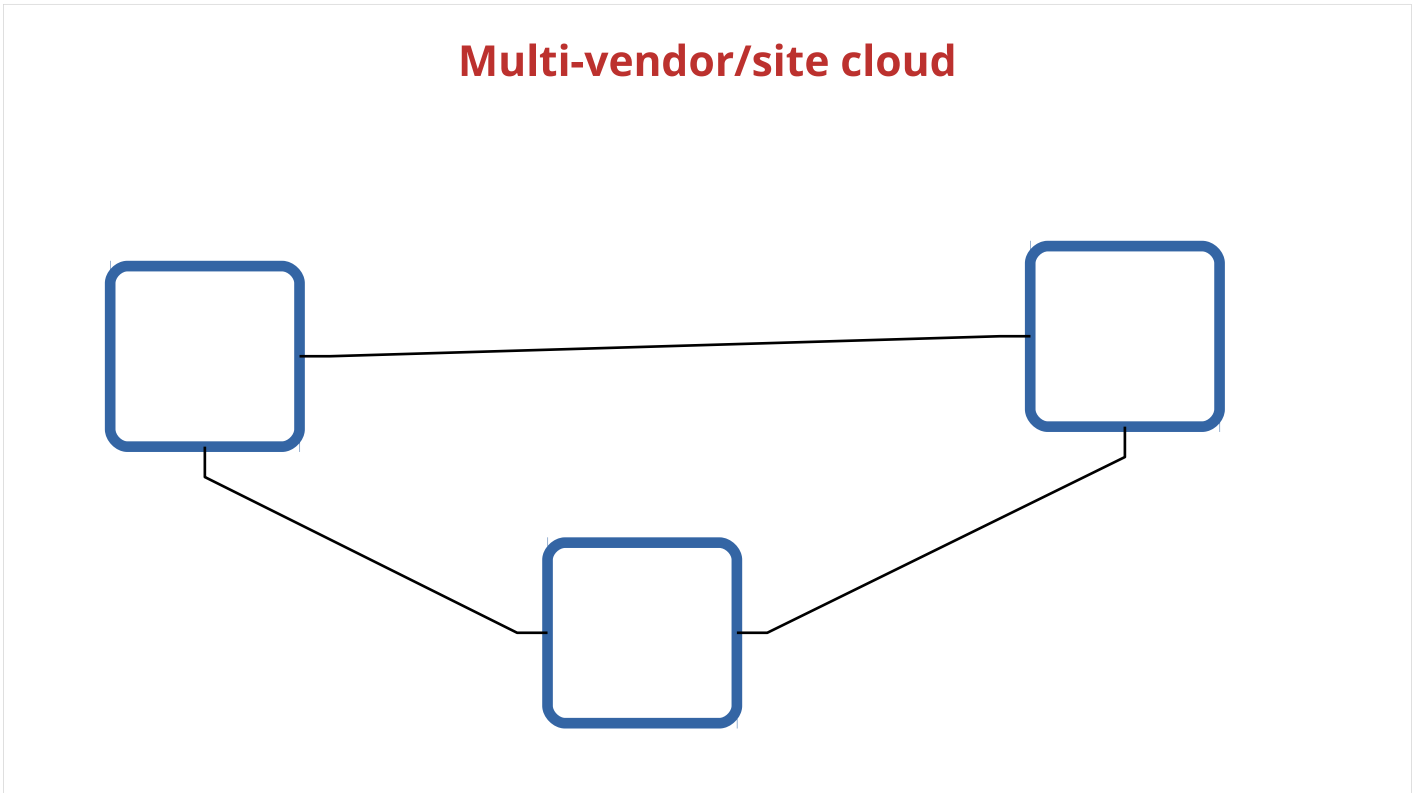


Each cluster acts as a standalone datacenter.

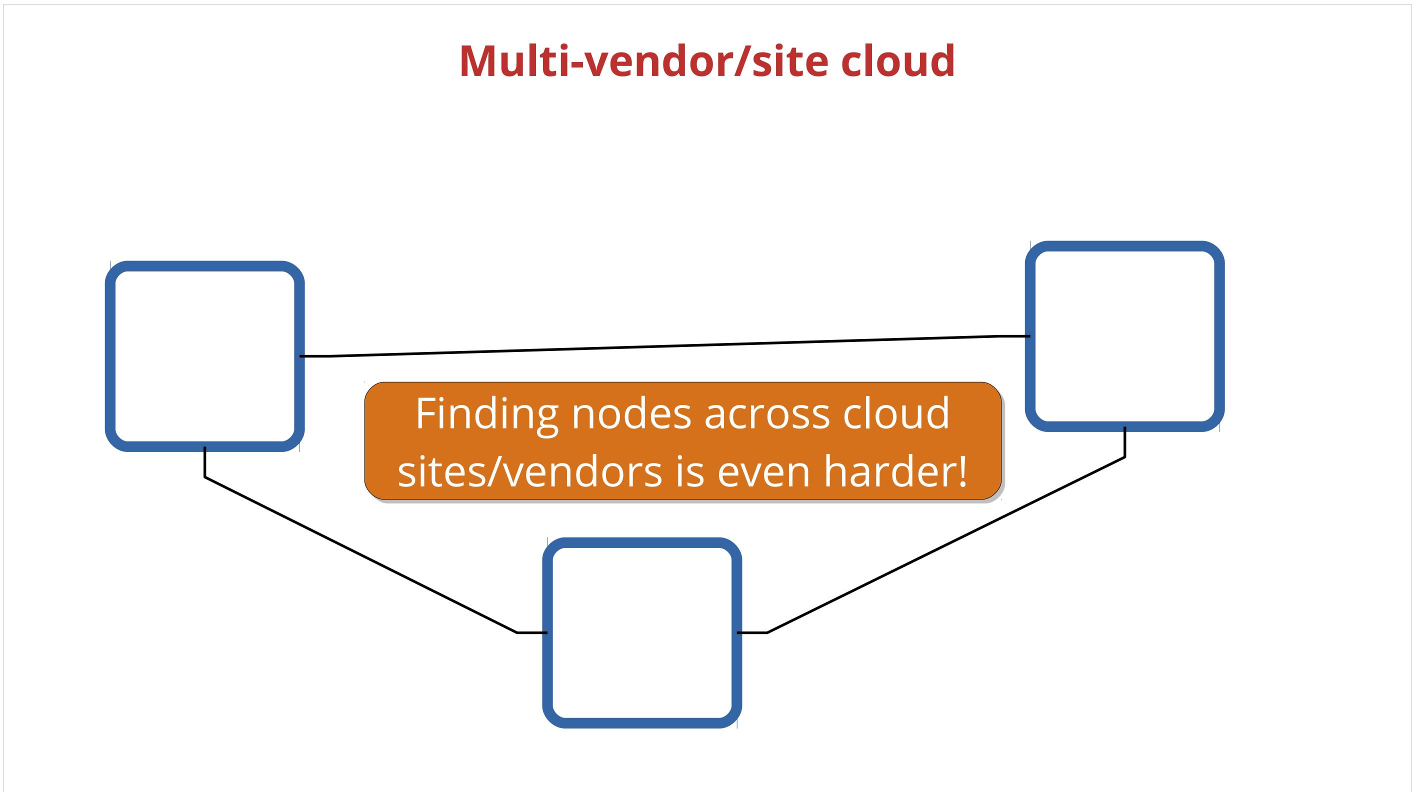
But this increases operational complexity.

.

We now, manage multiple datacenters instead of actually one.



Another challenge is when dealing with multi-site or multi-vendor cloud environments.



The problem is even worse in such environments. There are multiple players and it’s not longer the case where we have only one control plane.

Scalable and generic search service for distributed systems

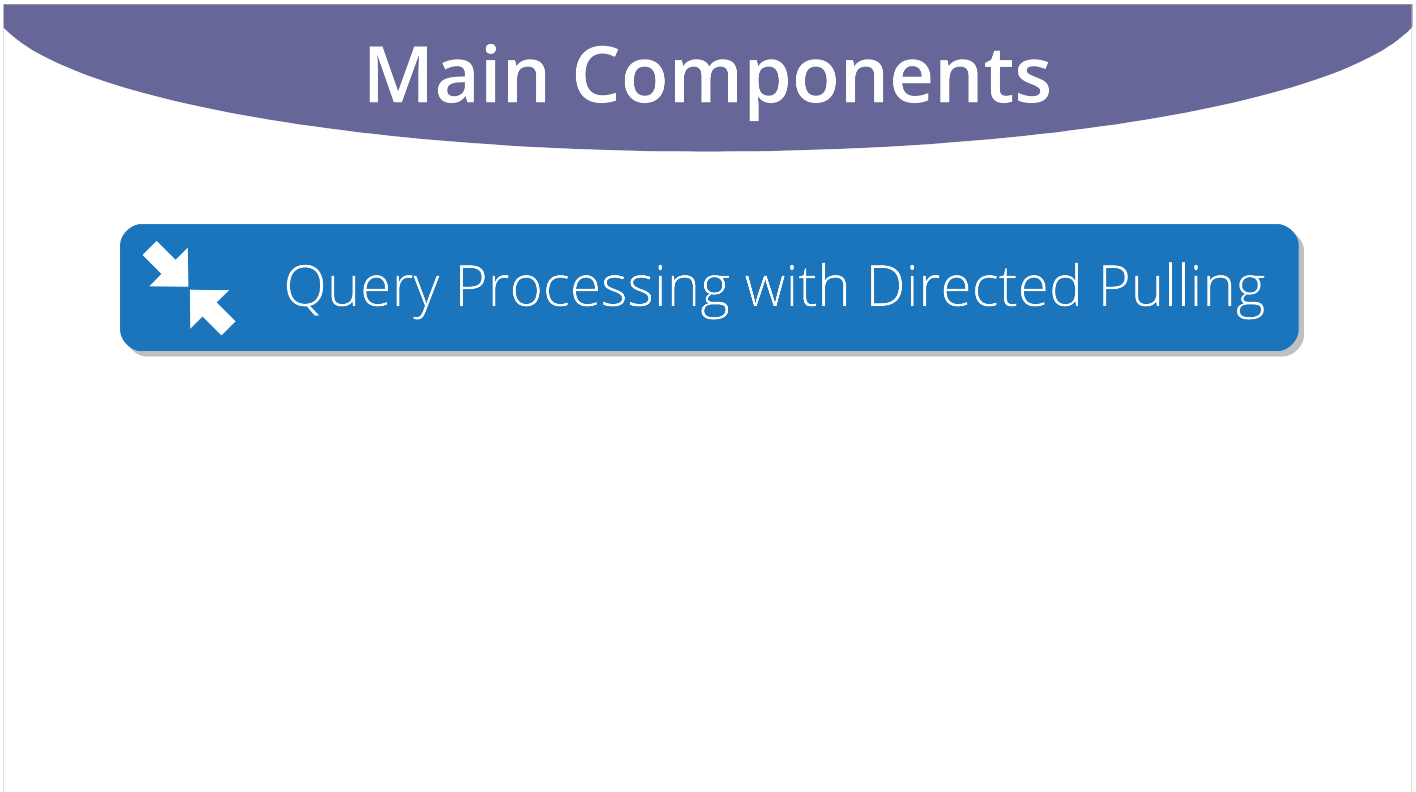
To that end, we present FOCUS:

.

A salable and generic search service for distributed systems.

Main Components

There are three main components or concepts that make up FOCUS.



First, we perform optimized querying with a concept we call directed pulling.

.

So, instead of querying all nodes in the system, we only query those that we think will have the potential to satisfy such queries.

Main Components

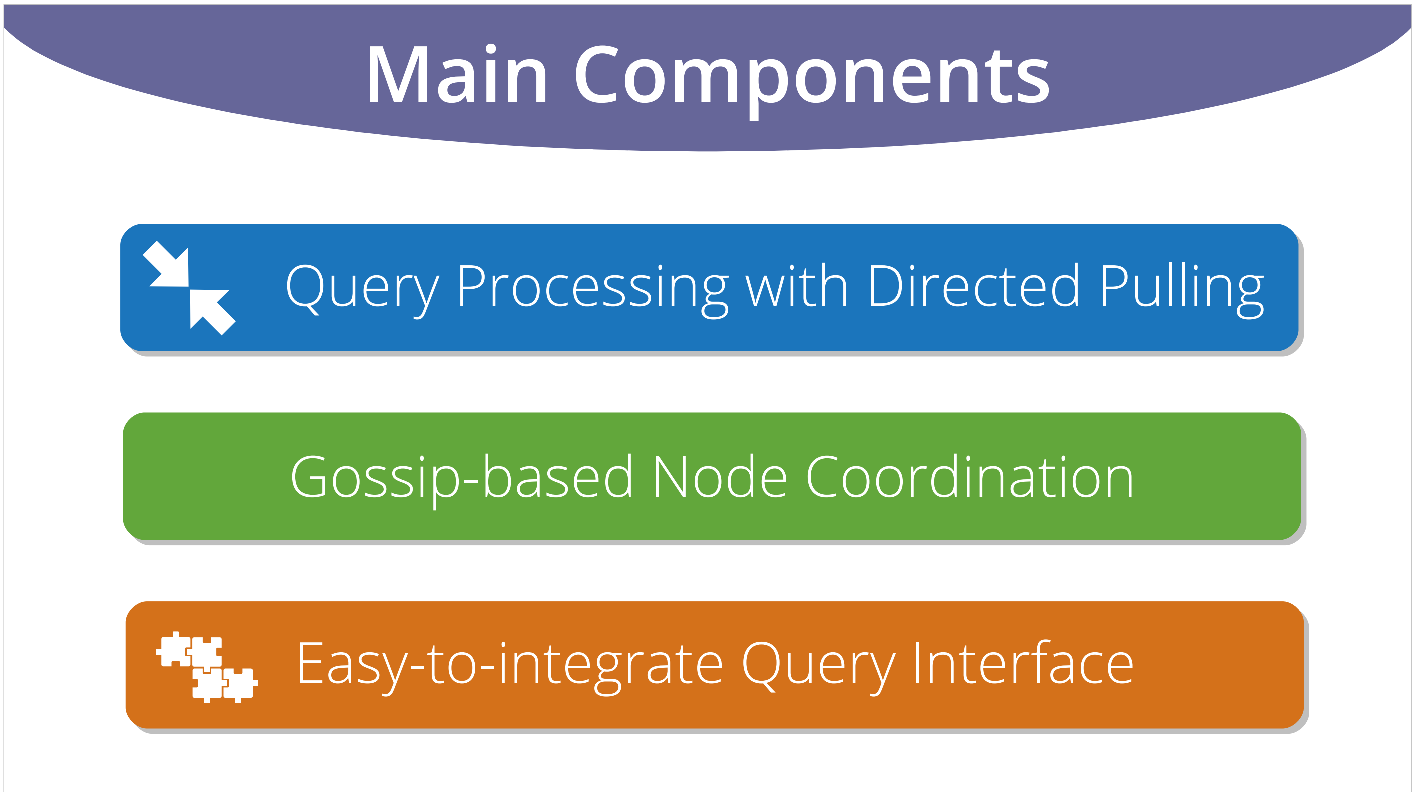
Query Processing with Directed Pulling

Gossip-based Node Coordination

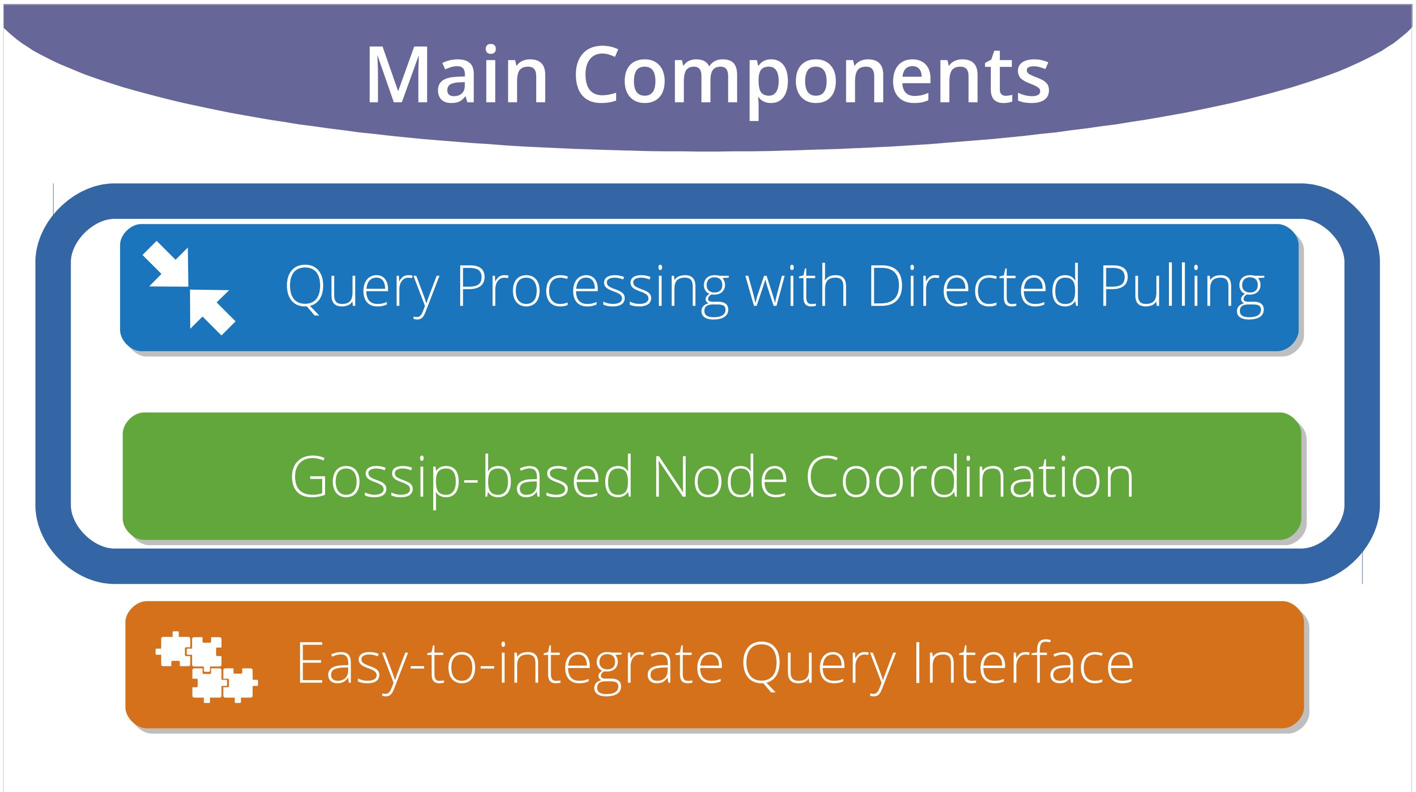
Another concept that we employ is that nodes in the system connect to one another and manage themselves.

.

This form of decoupling allows us to reduce the load on a centralized control entity, and thus scale much better than current approaches.



Lastly, we design the system in a way that allows it to easily integrate with existing systems through providing a neutral API, and we actually show how it can be done by integrating with OpenStack.



However, due to the limited time, I will only go over the first two concepts, and you can find details on our integration with OpenStack in section 9 in the paper.

Abstractions

First, I’ll describe some of the important abstractions we made.

Abstractions

Node Attributes

Each node in the system has some attributes.

Abstractions

Node Attributes

- Static
- Dynamic

Which can be divided into either static or dynamic

Static attributes never change

Abstractions

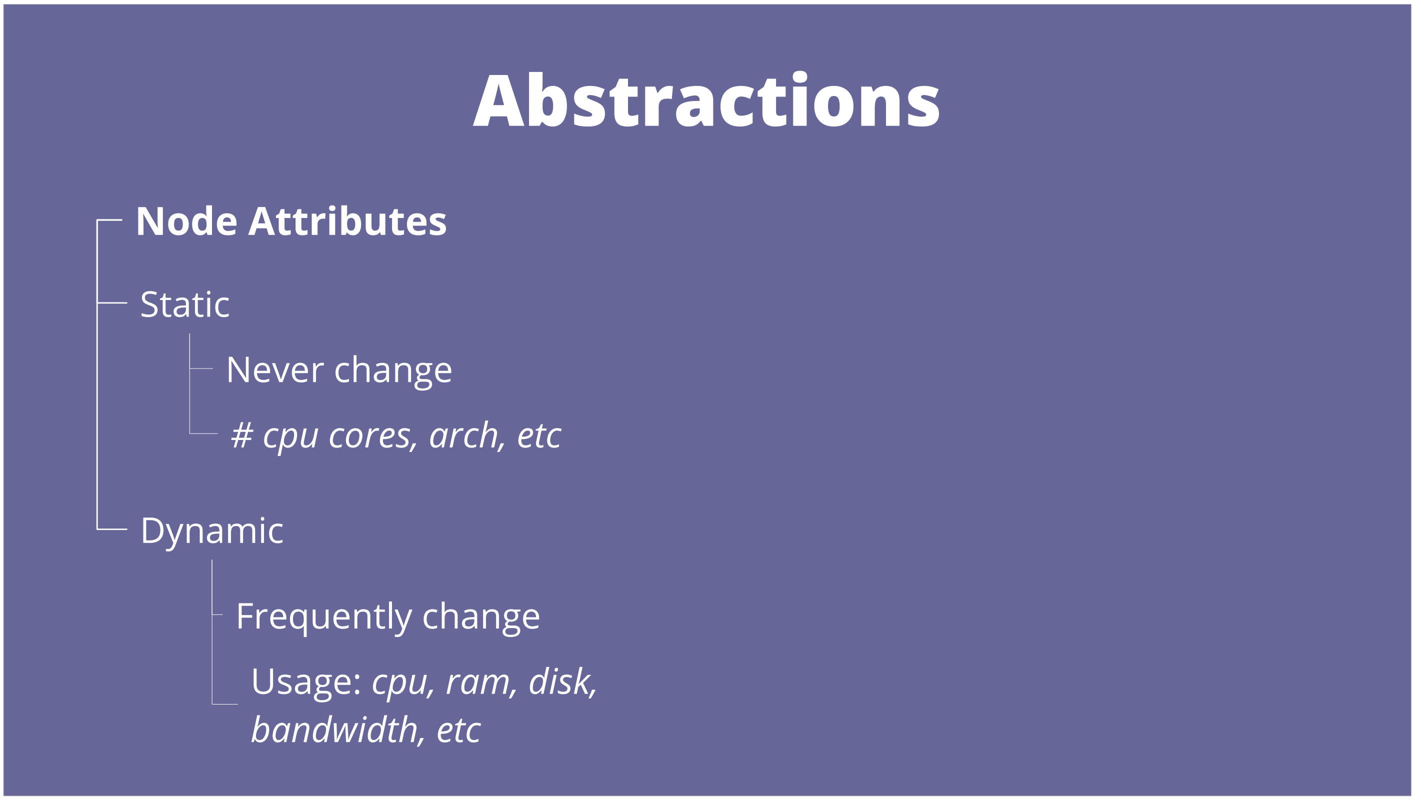
- **Node Attributes**
- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic

For instance, the number of physical cores, or the architecture of the machine are things that never change.

Abstractions

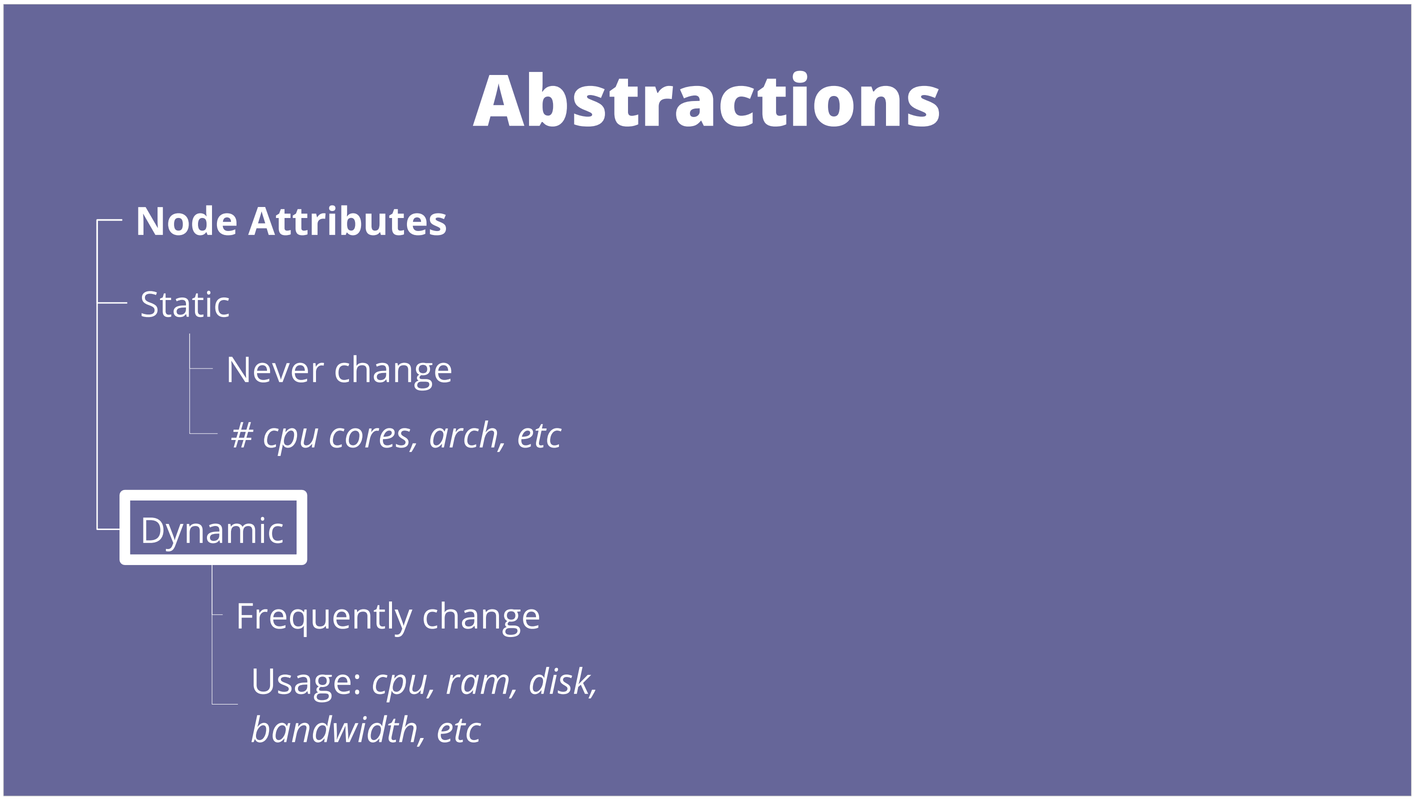
- **Node Attributes**
  - Static
    - Never change
    - *# cpu cores, arch, etc*
  - Dynamic
    - Frequently change

However, dynamic attributes DO change..
.
And they might change too frequently, depending on the workload and the environment.

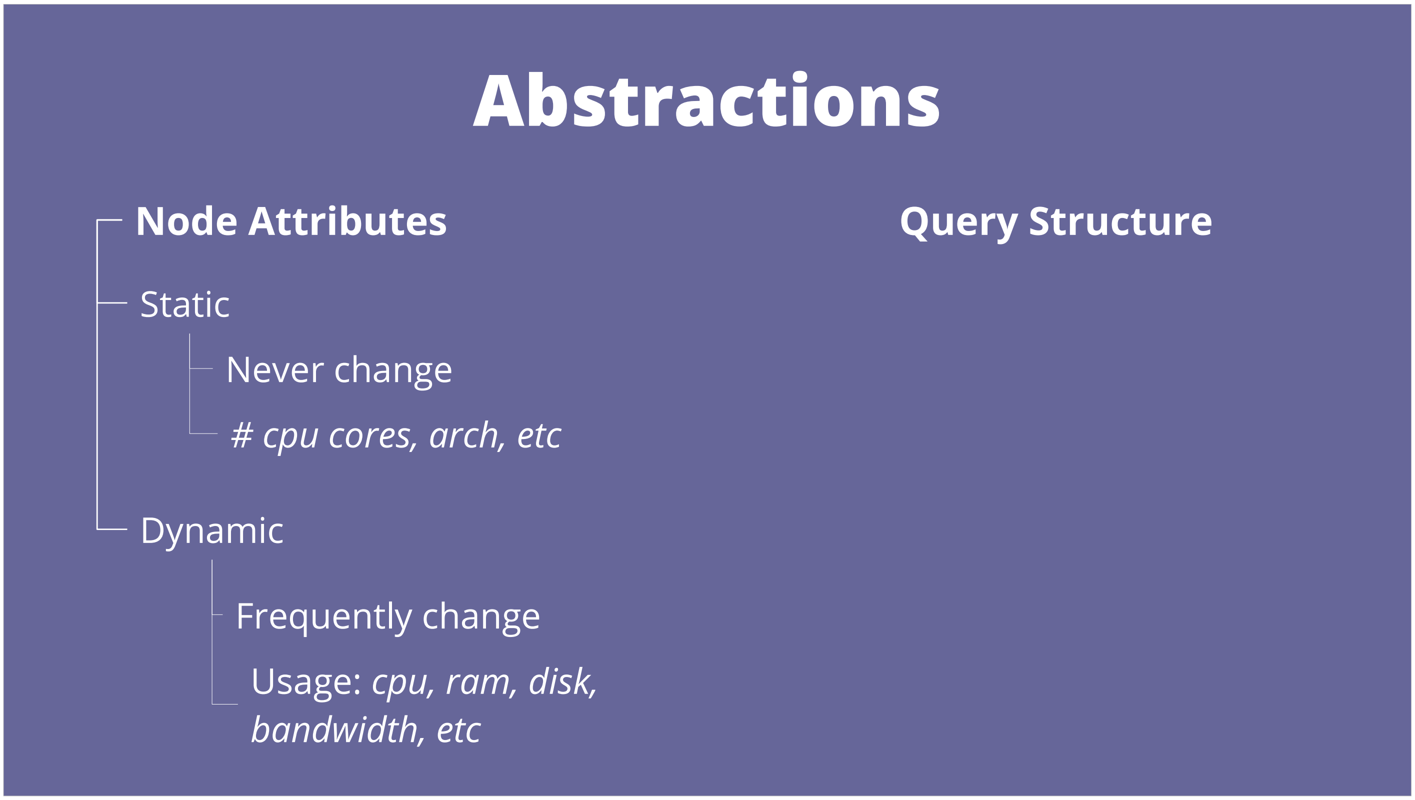


These are things like, usage information such as CPU, RAM, DISK, or bandwidth.

.



Since static attributes can be cached once and obtained anytime,
We focus on dynamic attributes, and design our system accordingly.



Another important abstraction is the Query structure.

Abstractions

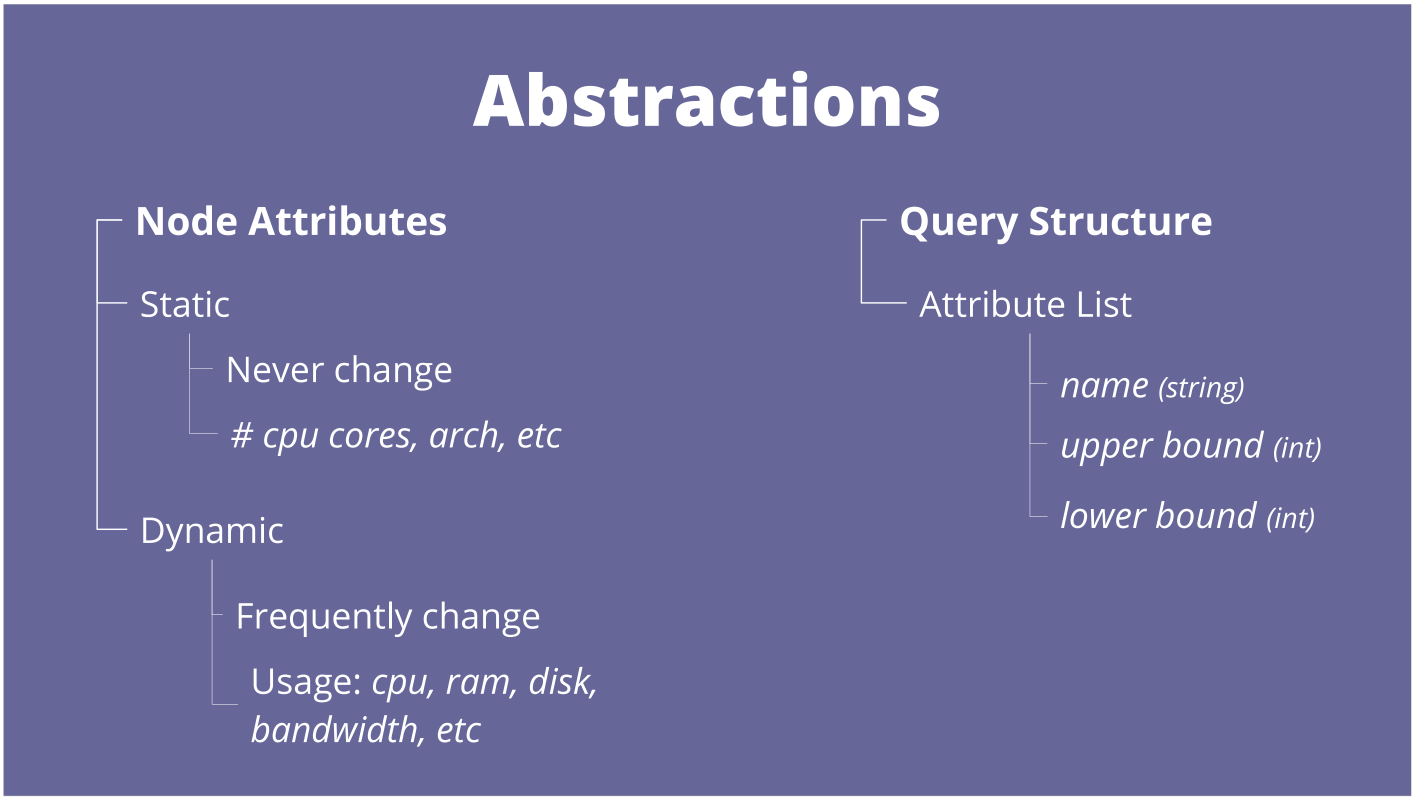
Node Attributes

- Static
  - Never change
  - *# cpu cores, arch, etc*
- Dynamic
  - Frequently change
  - Usage: *cpu, ram, disk, bandwidth, etc*

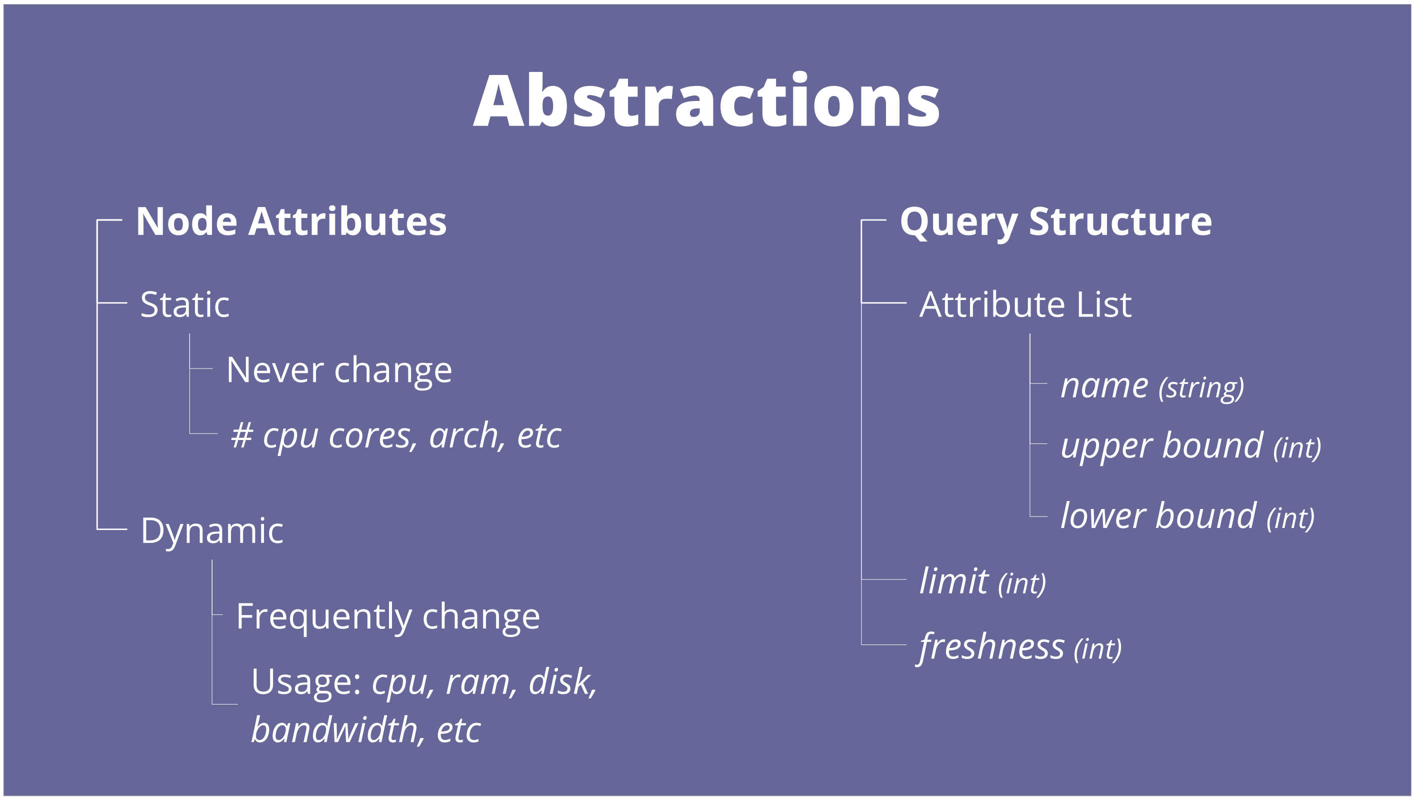
Query Structure

- Attribute List

Each query will have a list of attributes and their desired values.



Which is made of three fields:
1- name of the attribute.
2- upper bound of the value
3- and lower bound of the value



Two other parameters are: limit and freshness of the results.

.

You can specify a limit on the number of returned results because if the limit is smaller, then we can return results in a much shorter time.

.

And you can specify how fresh the results should be. If the query does not require fresh results, then we can always get cached information to boost the performance.

Query Processing with Directed Pulling

Ok, so how can we send queries to only those nodes that we “think” will have the potential to satisfy the query?

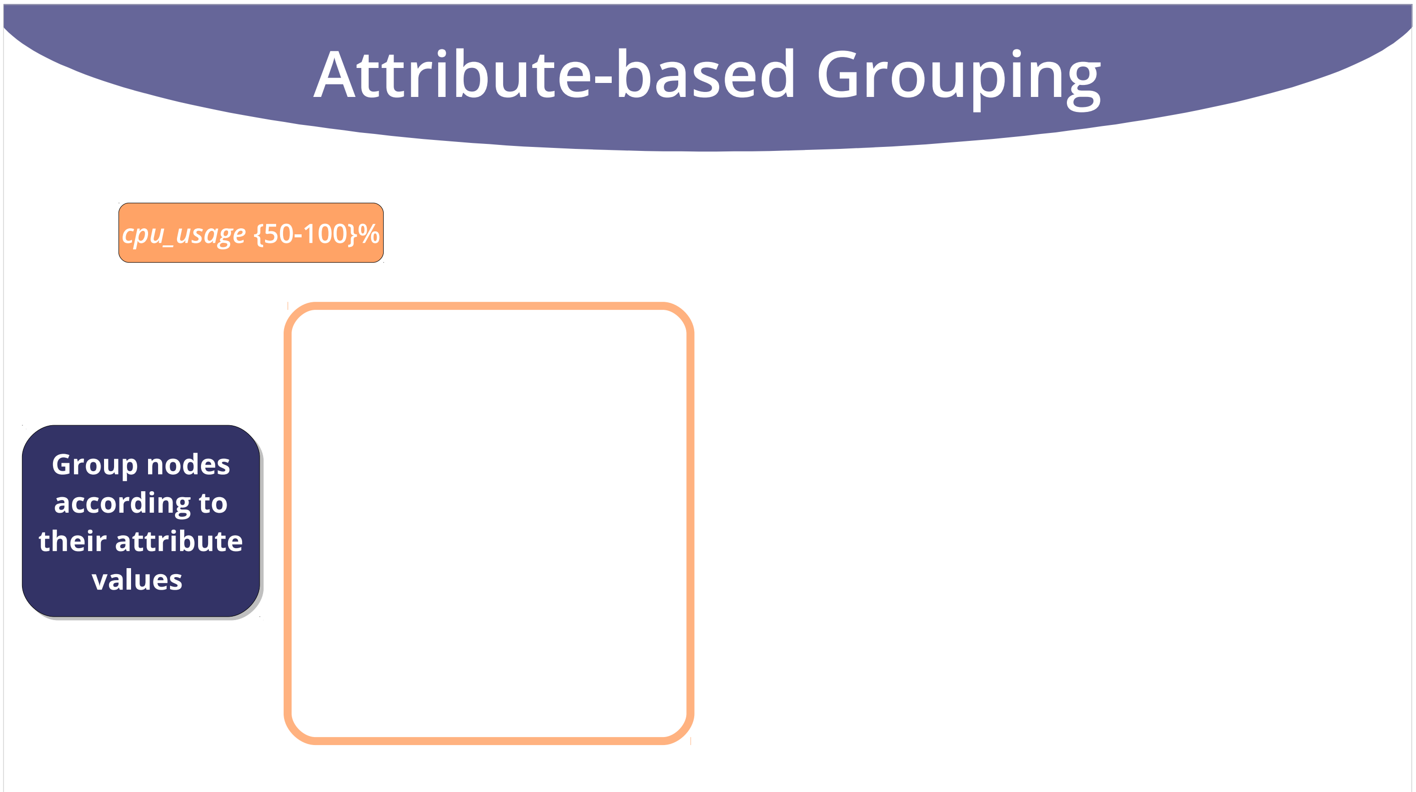
Attribute-based Grouping

We achieve that by using a novel concept we call: attribute-based grouping.

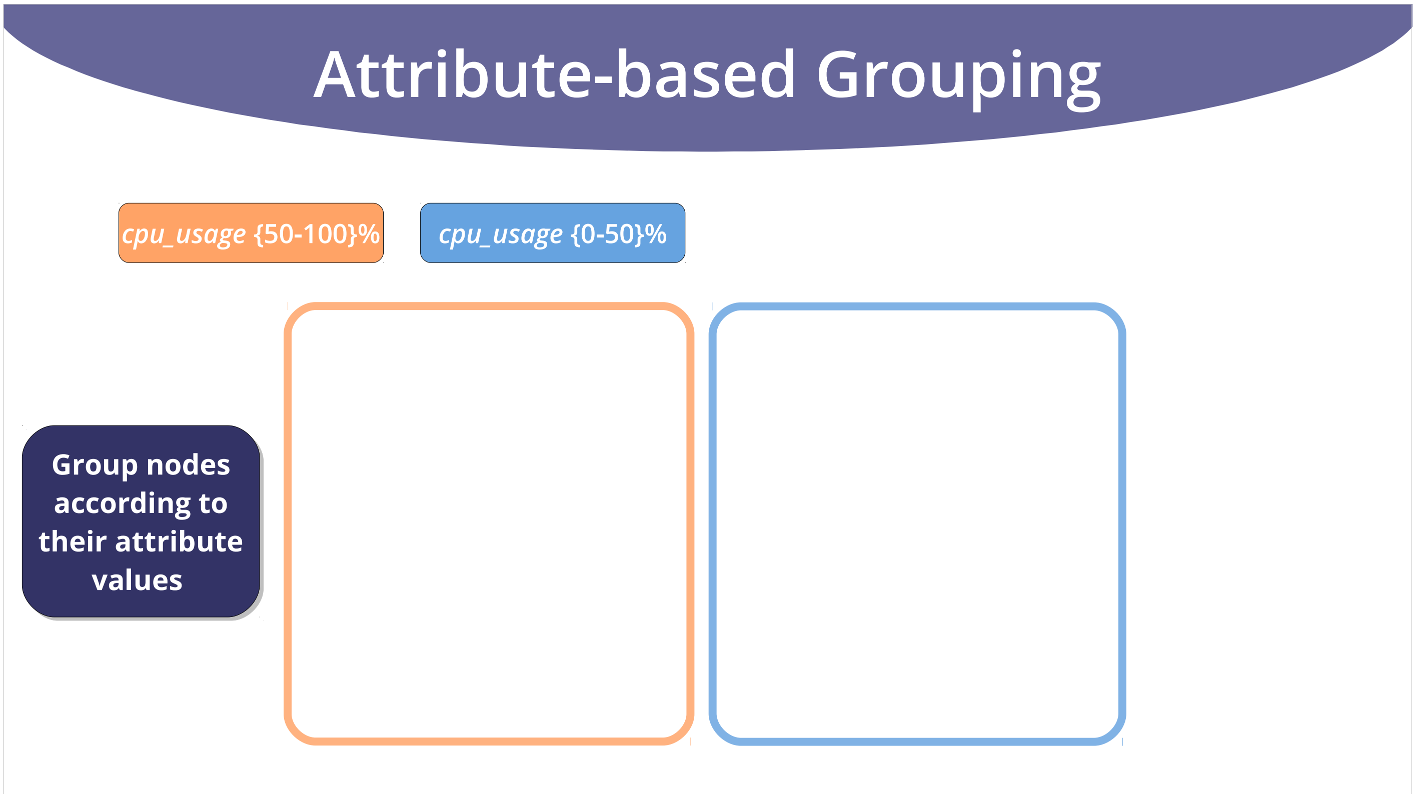
Attribute-based Grouping

Group nodes according to their attribute values

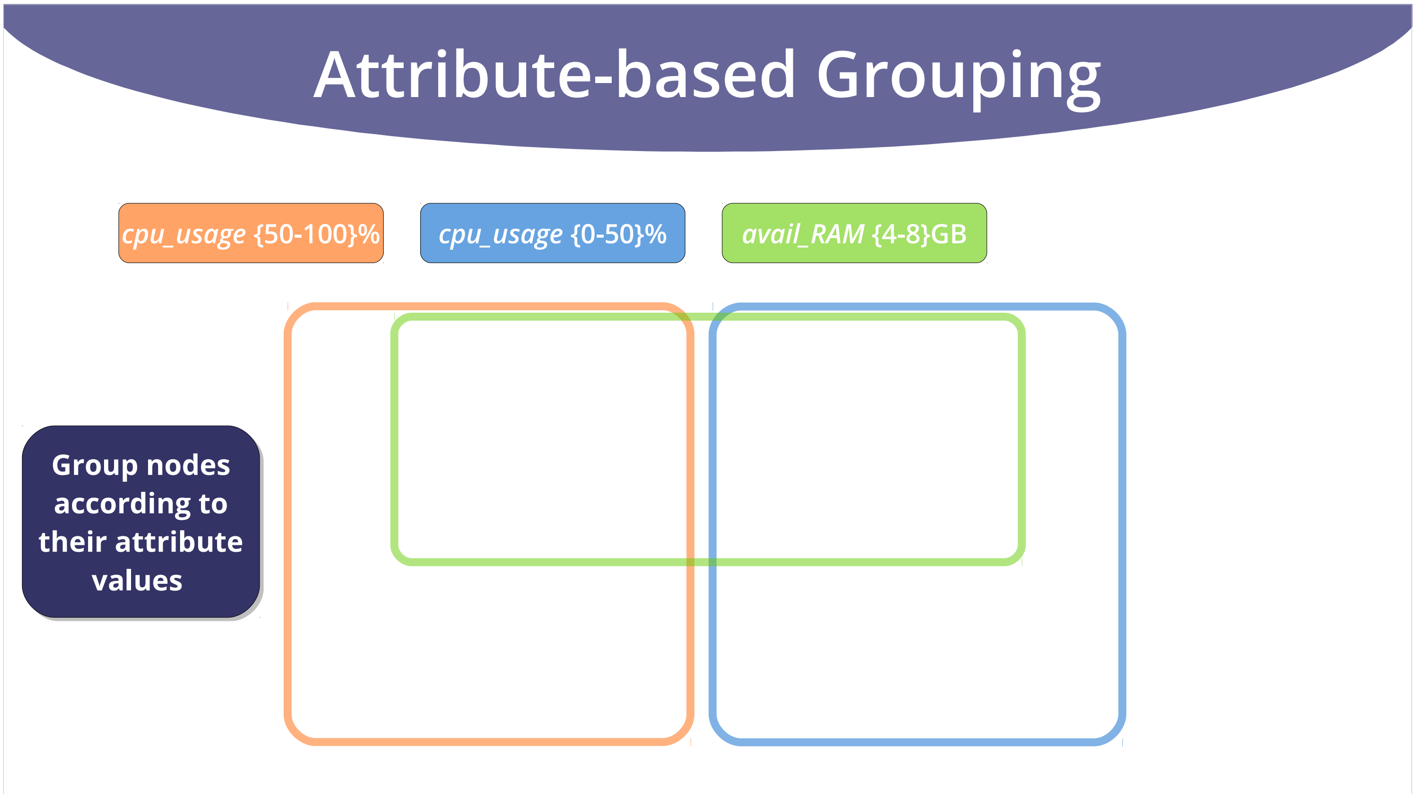
That is, we group nodes that have similar attribute values together.
One group per attribute value range.



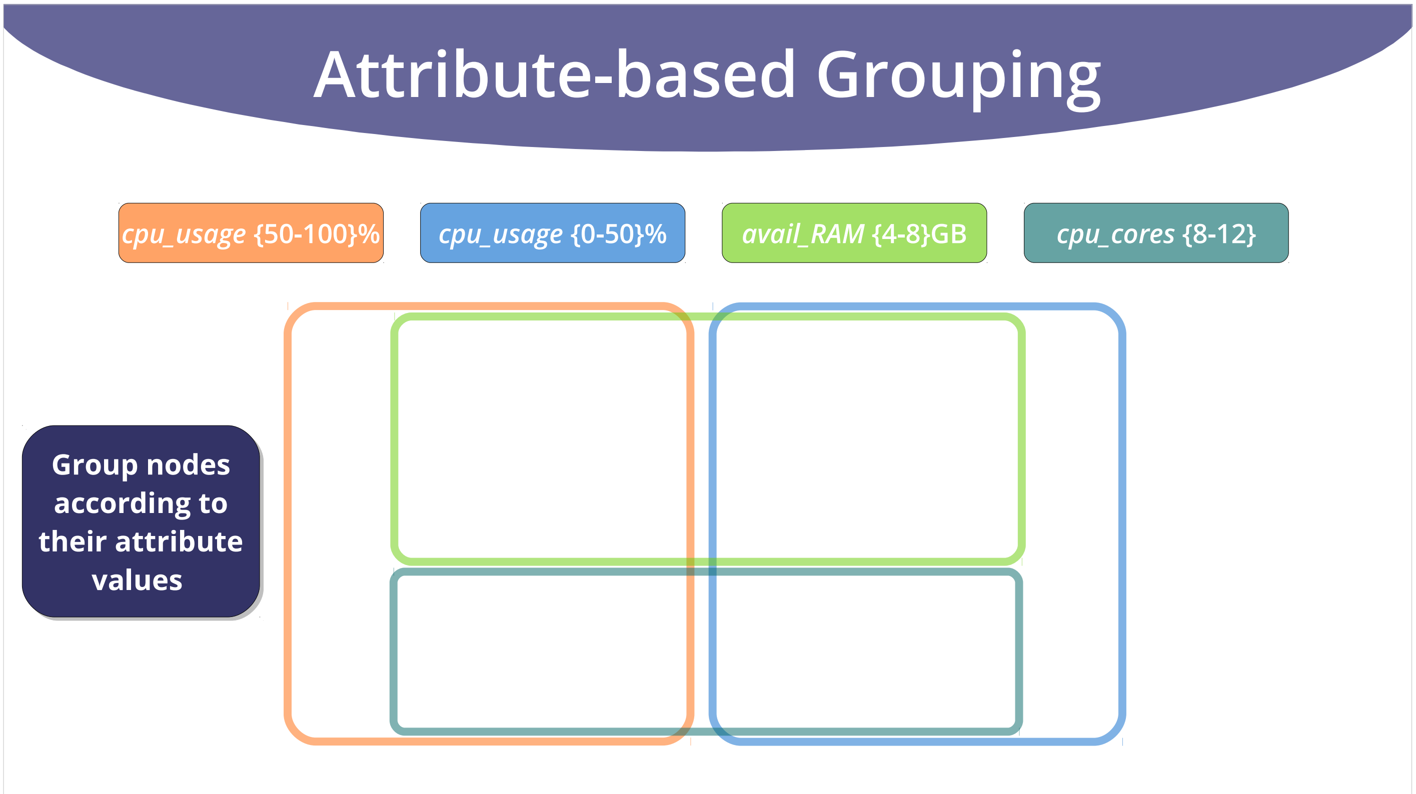
For instance, we might want to group nodes that have a cpu usage above 50%.



And another group for the other nodes.



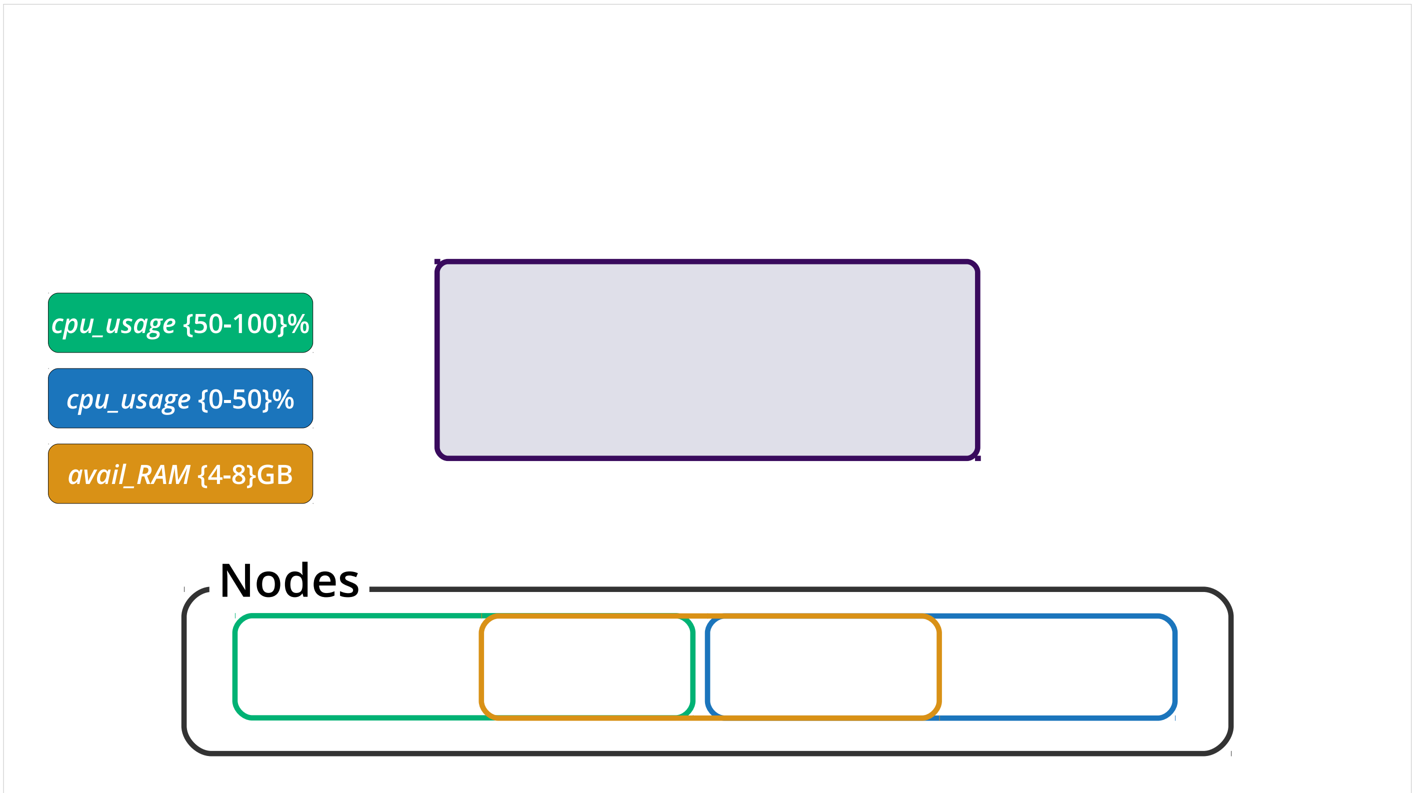
And for other attributes, we create separate groups.



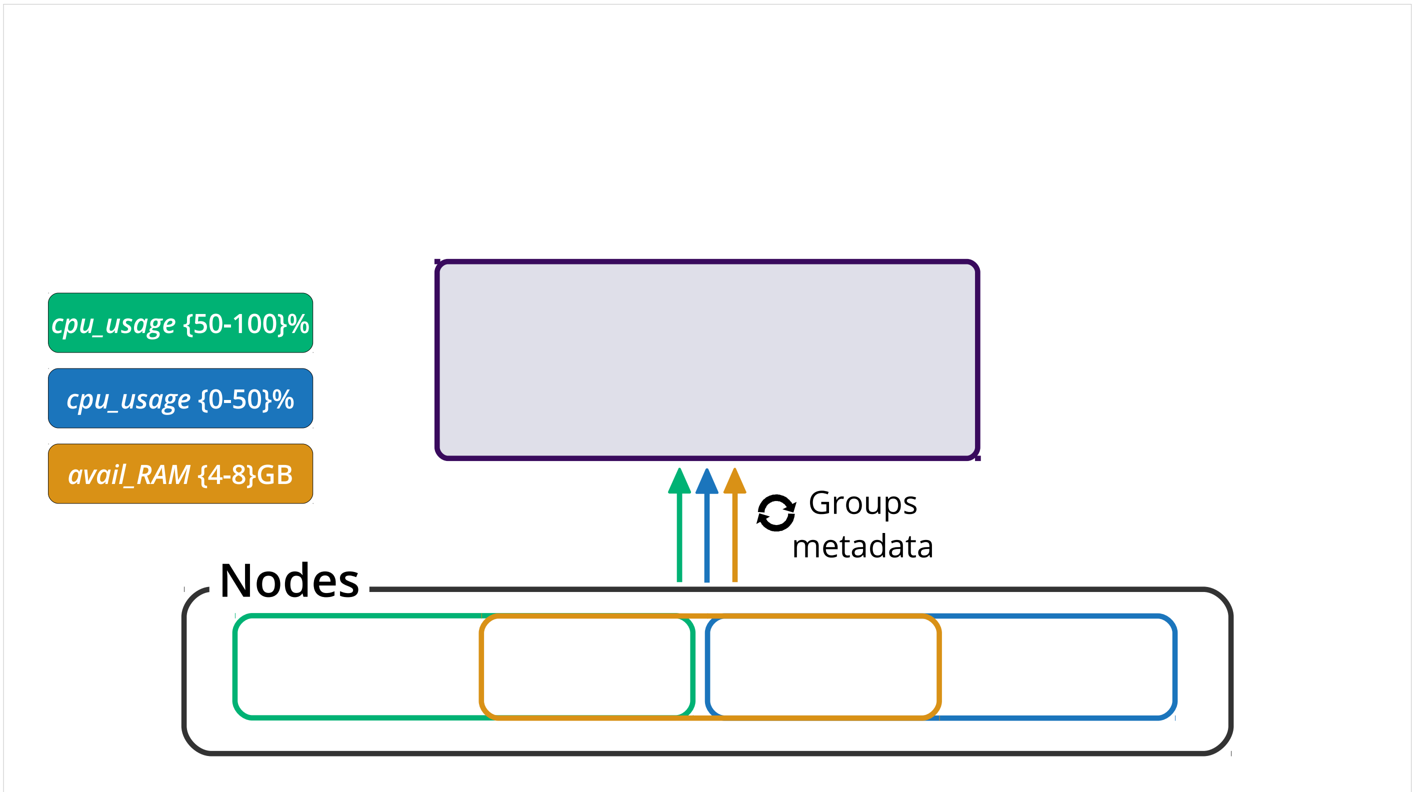
And so on.

.

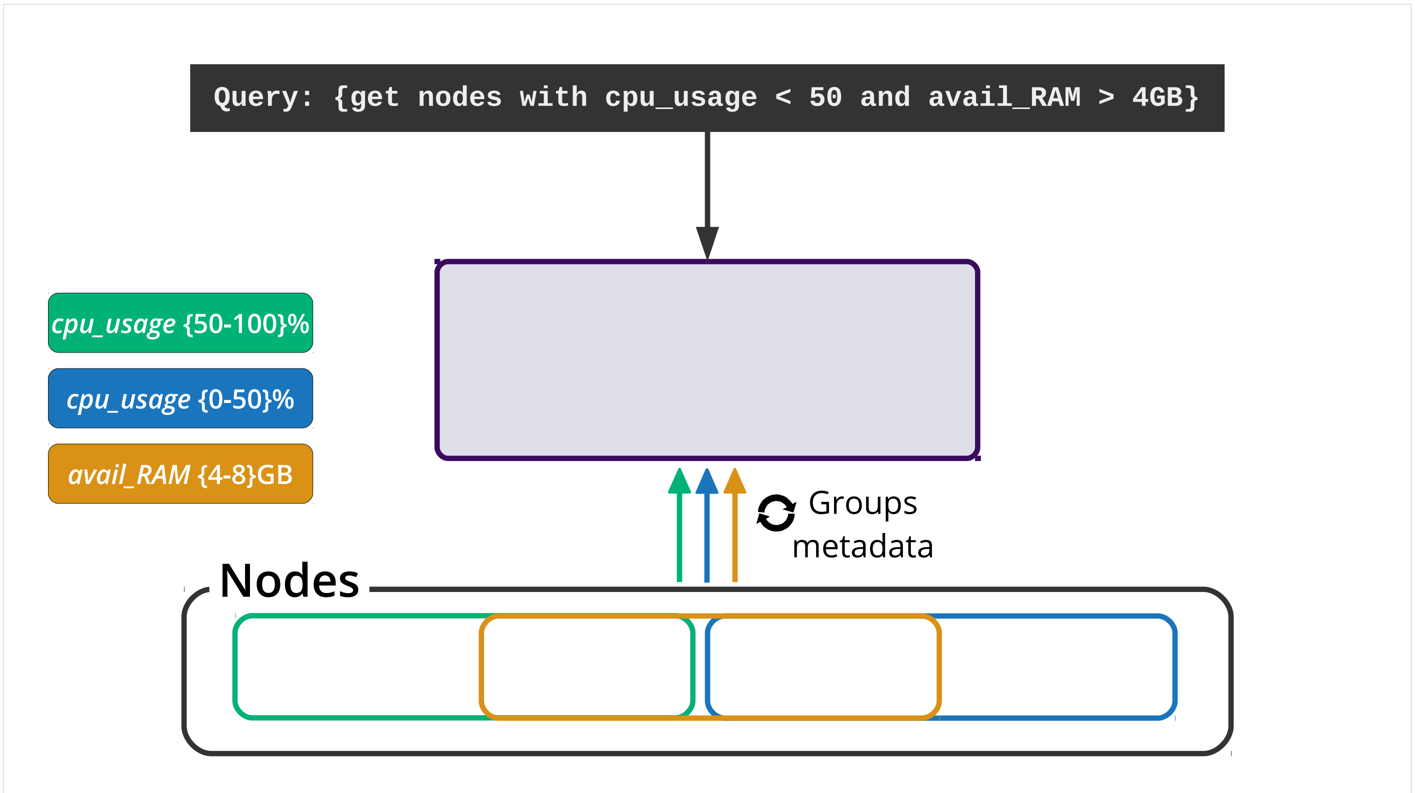
A key thing here is that a node can be in X number of groups if it has X number of attributes.



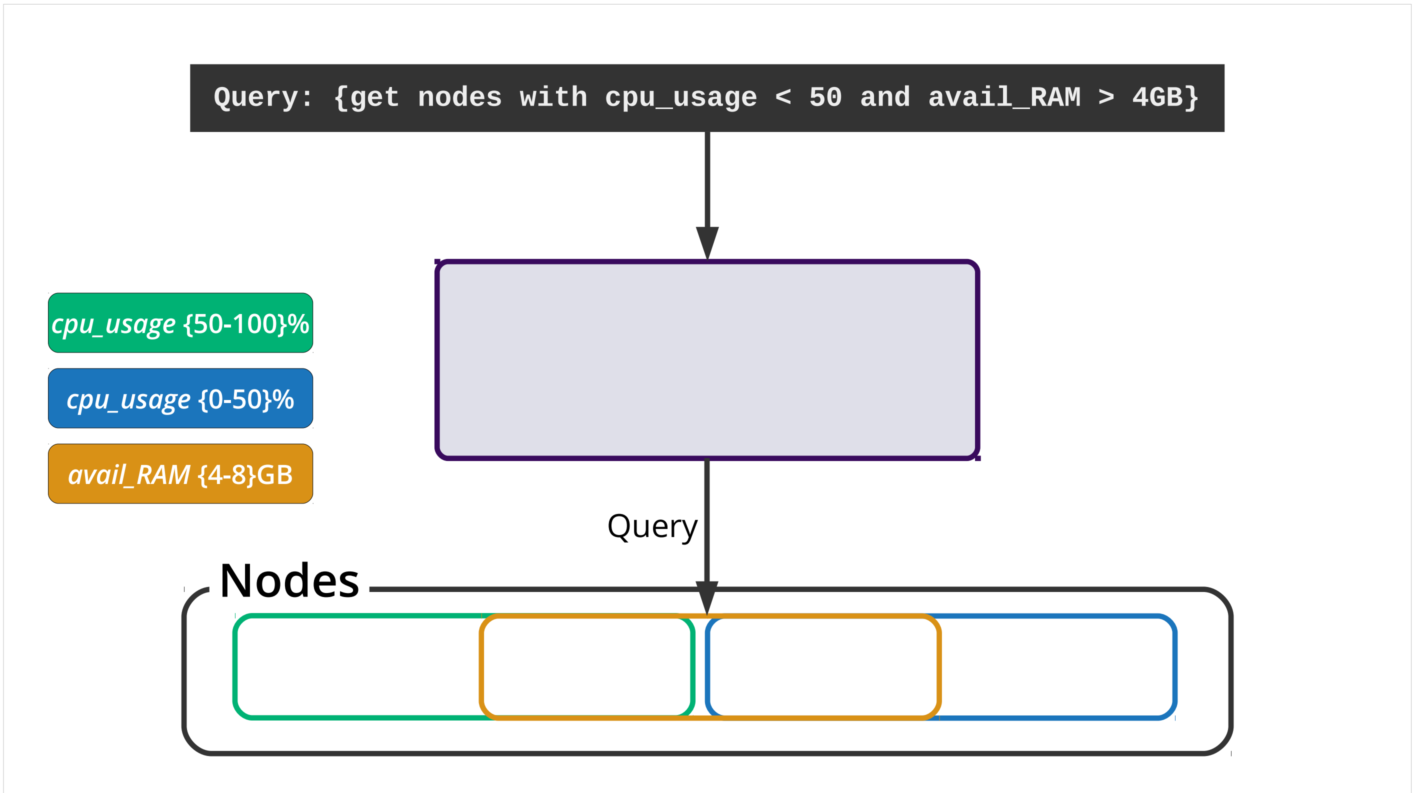
Now that we logically organized nodes into different groups based on their attribute values, how can we actually tell which nodes are in which group?



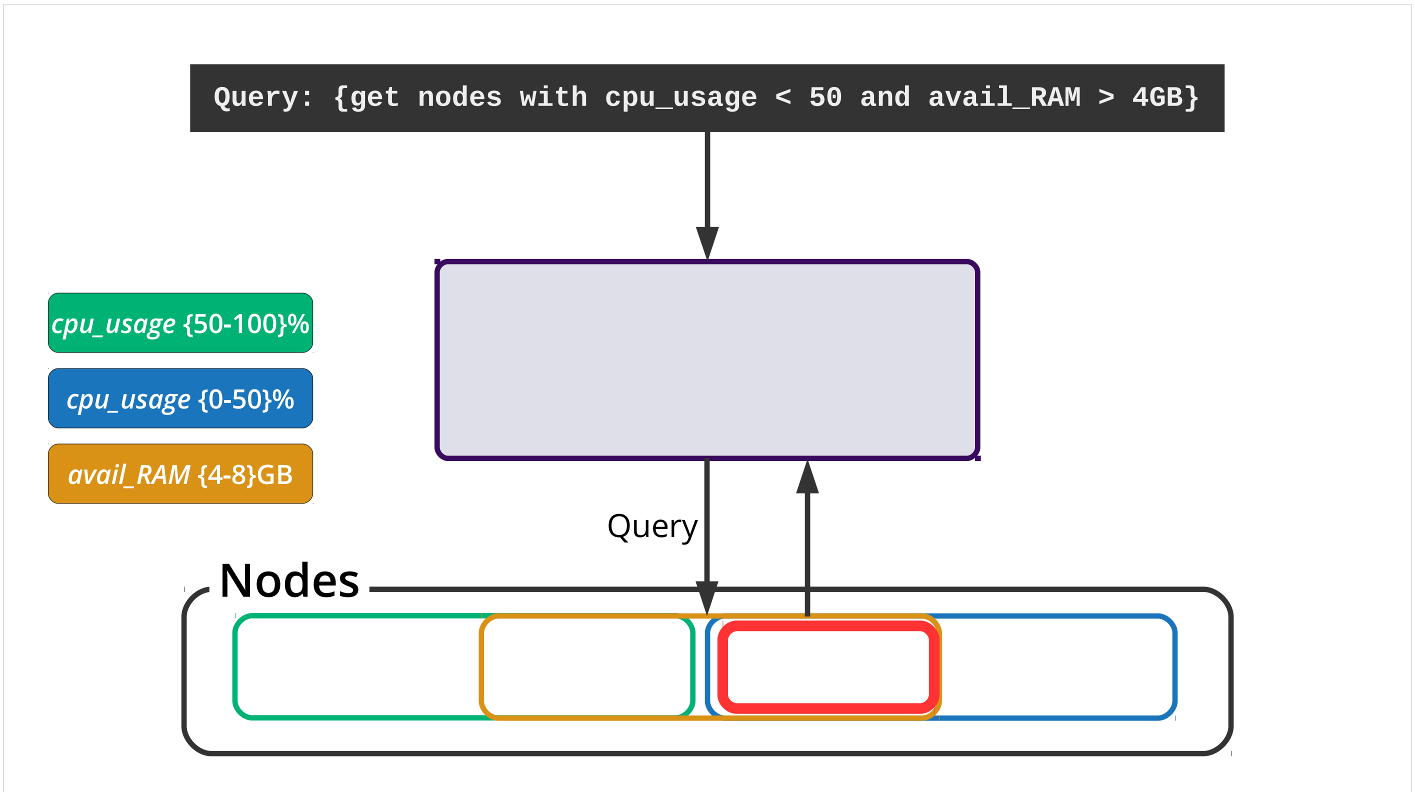
For each of those groups, we select a node that periodically pushes group-wide information (containing membership information) to our service.



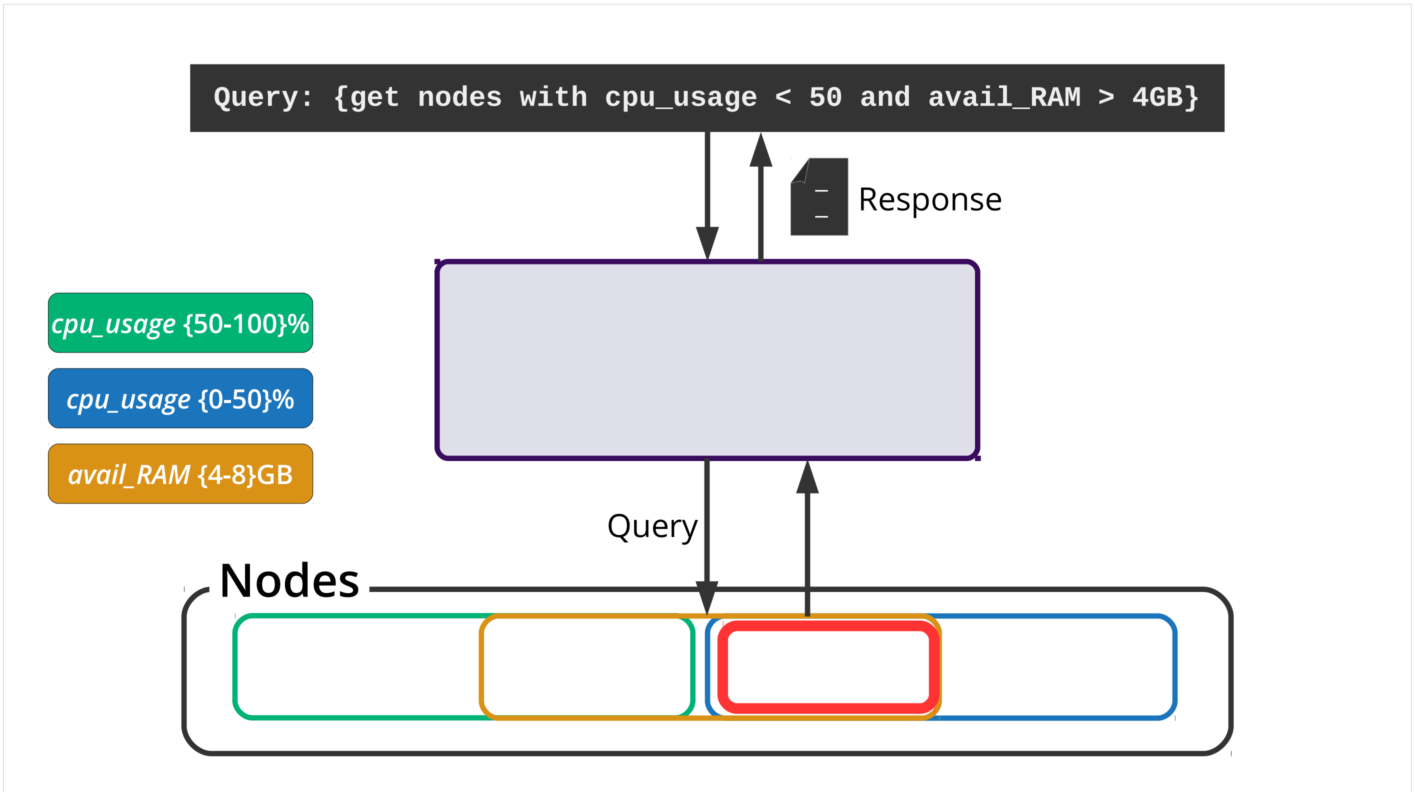
Then whenever we receive a query like this,



Acting on the group-wide information we periodically get, we send the query only to the group that satisfies the query.



In this case, we get up-to-date responses from nodes in the group.



And return a list of nodes that actually satisfy the query.

.

So, attribute-based grouping allows us to know in-advance which nodes are part of which group, and hence, which nodes have which information.

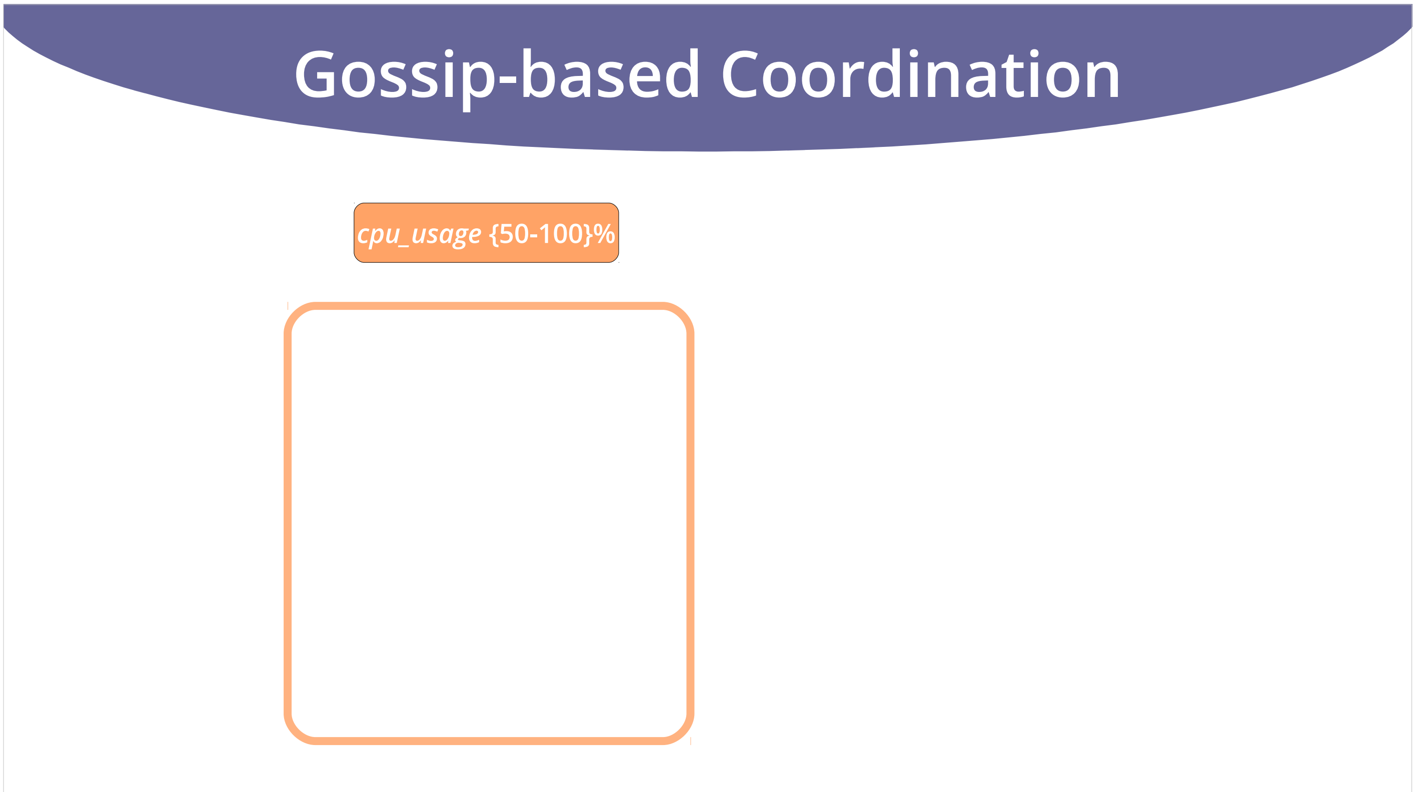
All of this is done in a dynamic and loosely-coupled manner.

Gossip-based Node Coordination

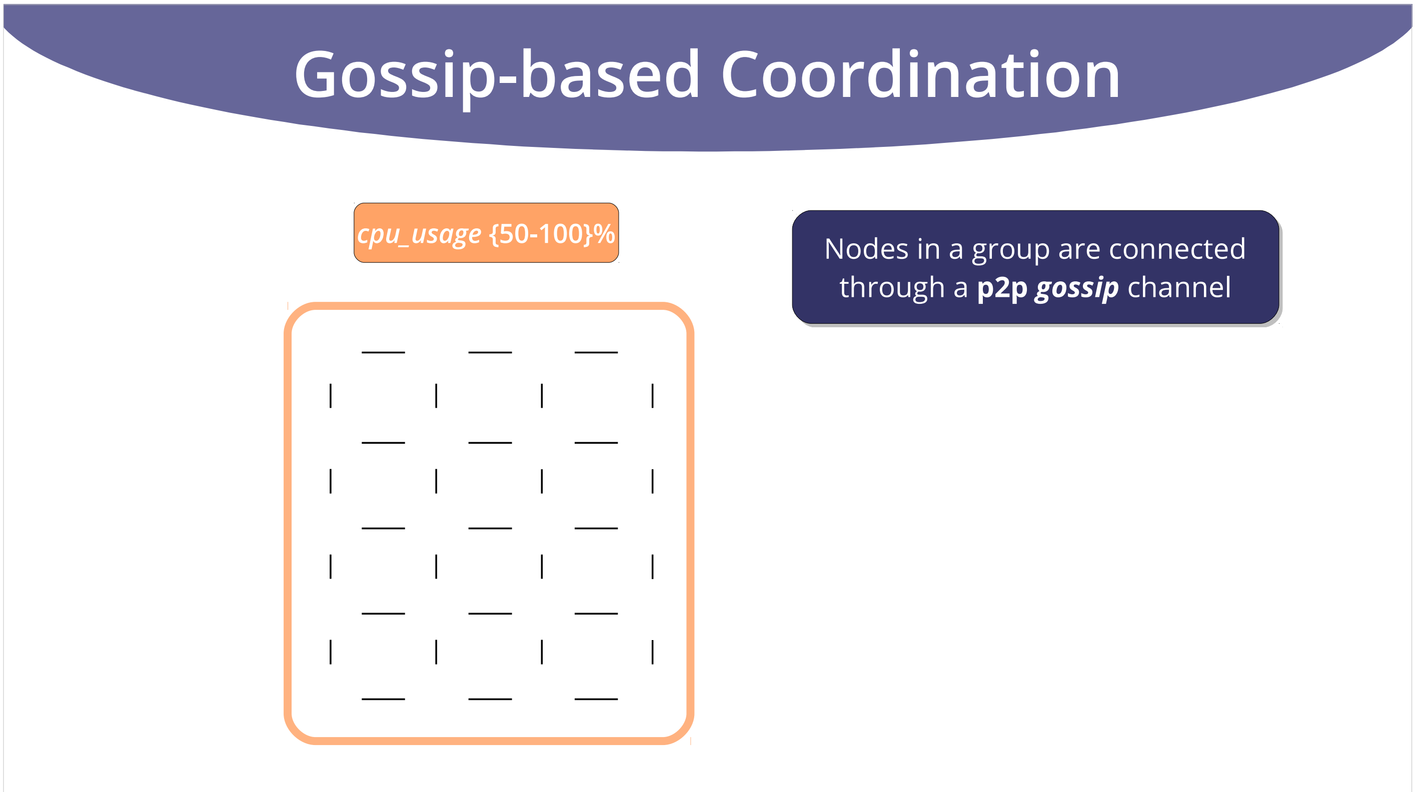
Ok, the second concept that motivates the design of FOCUS is that those groups are self-formed by the nodes.

.

And to coordinate how that's done, we let nodes form p2p groups that use the lightweight gossip-protocol to exchange messages.



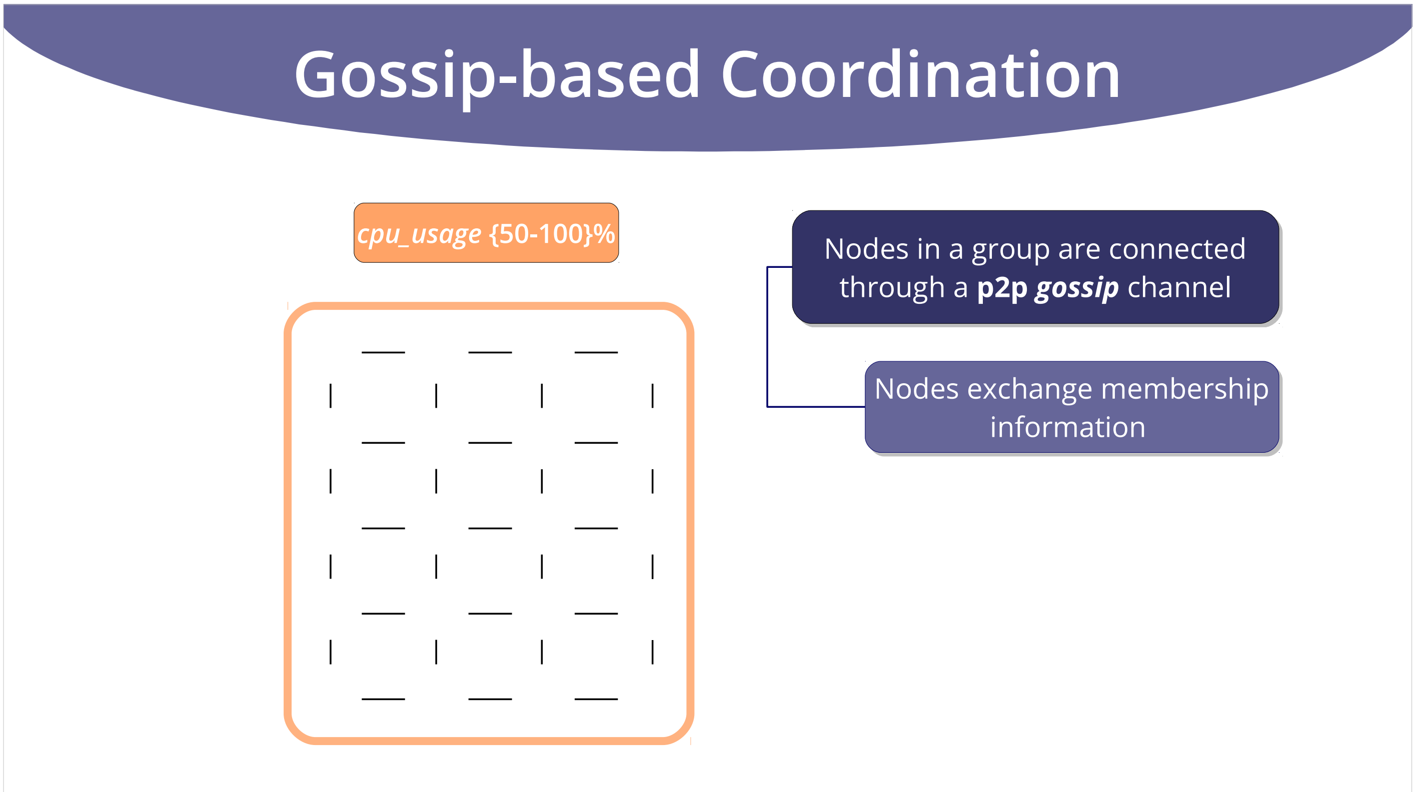
So, revisiting the example from earlier, here we have a group for nodes with CPU usage larger than 50%.



Each node of the group connects to only a few other members in the group.

.

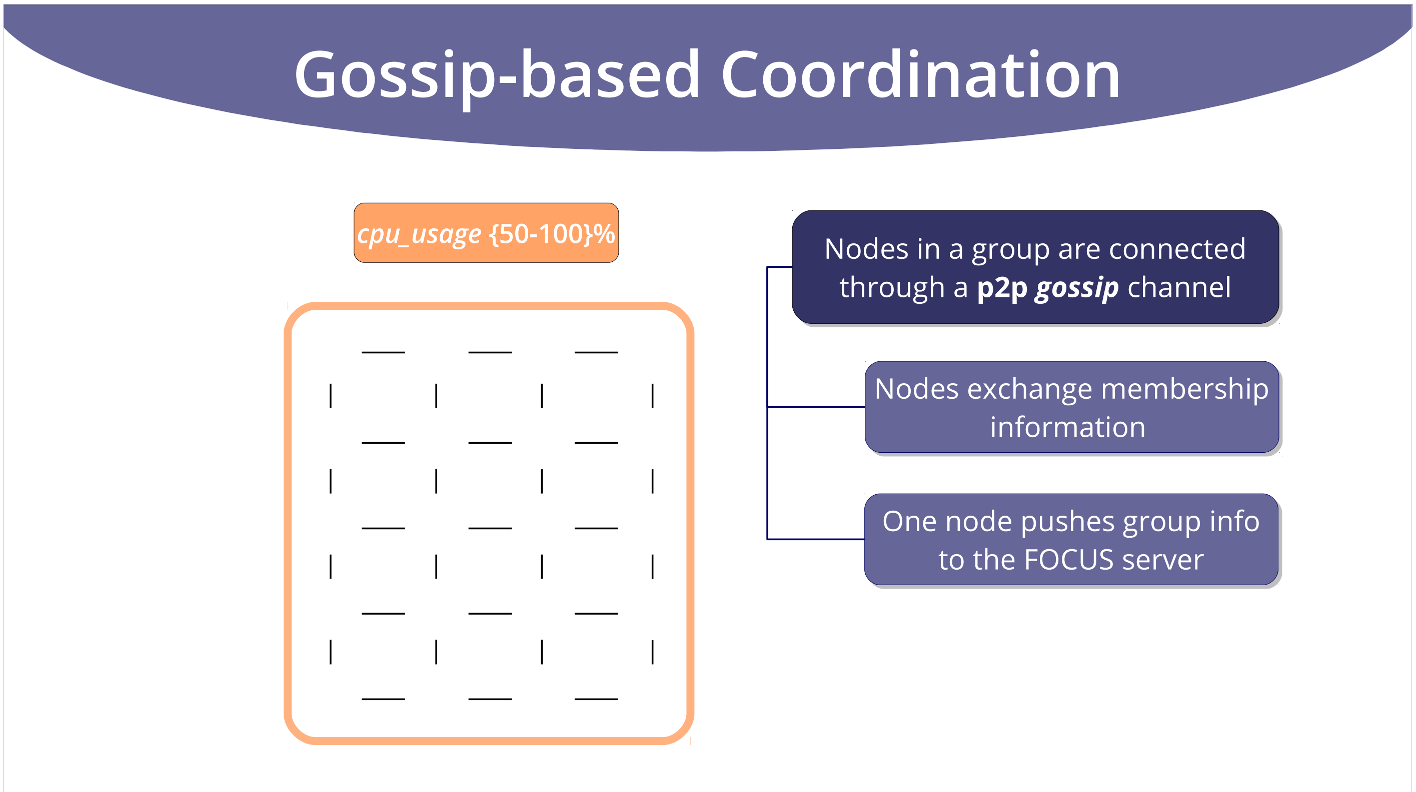
The number of peer nodes is referred to as the “fanout” parameter, which can be configured at start time.



Nodes within the same gossip group exchange messages with one another.

.

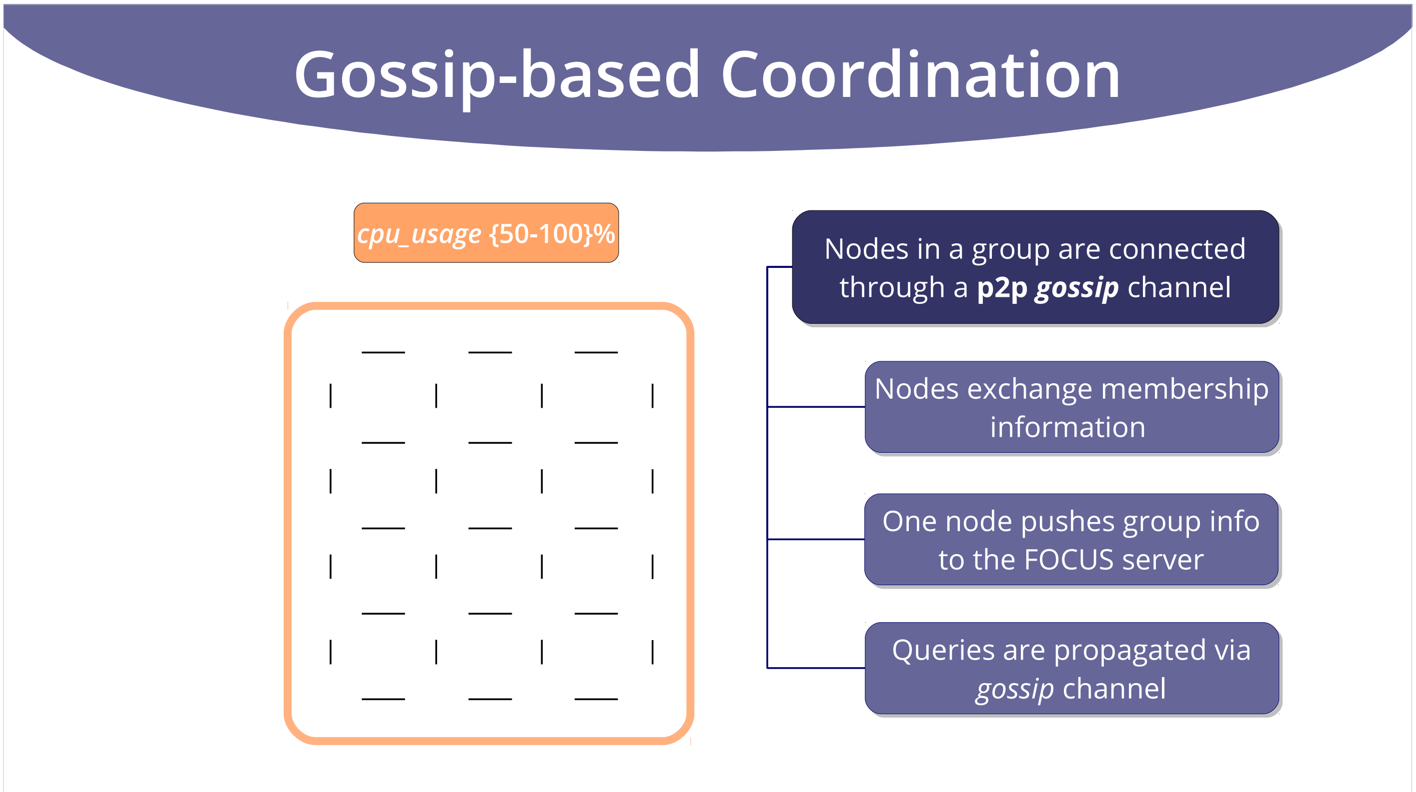
Such messages are mainly membership messages (telling who’s a member of the group)



We select one node in the group to be a group representative.

.

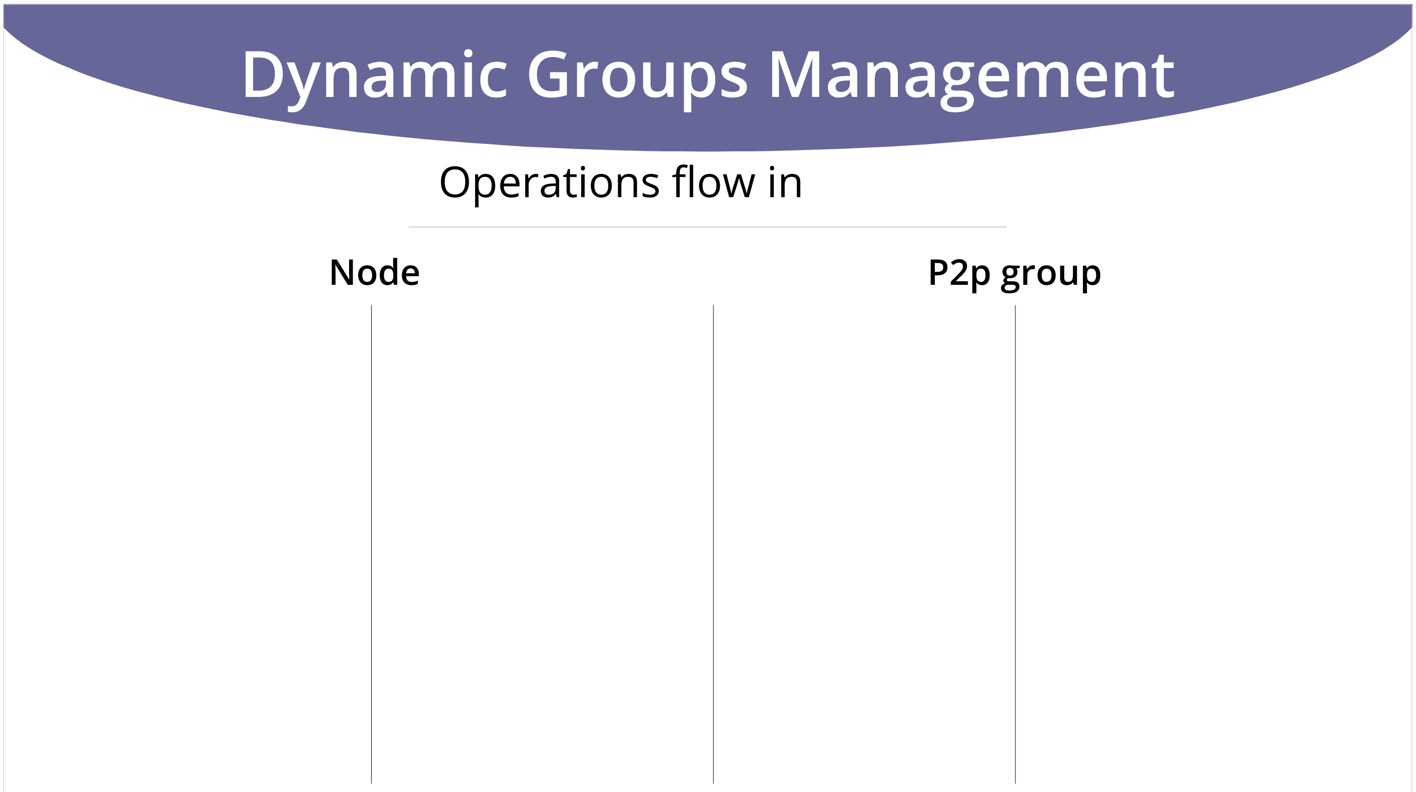
Its main job is to periodically synchronize the membership list with the FOCUS server.



Then whenever the FOCUS server receives a query, it will forward it to the group whose metadata (i.e., the group attribute value range) satisfies the query.

.

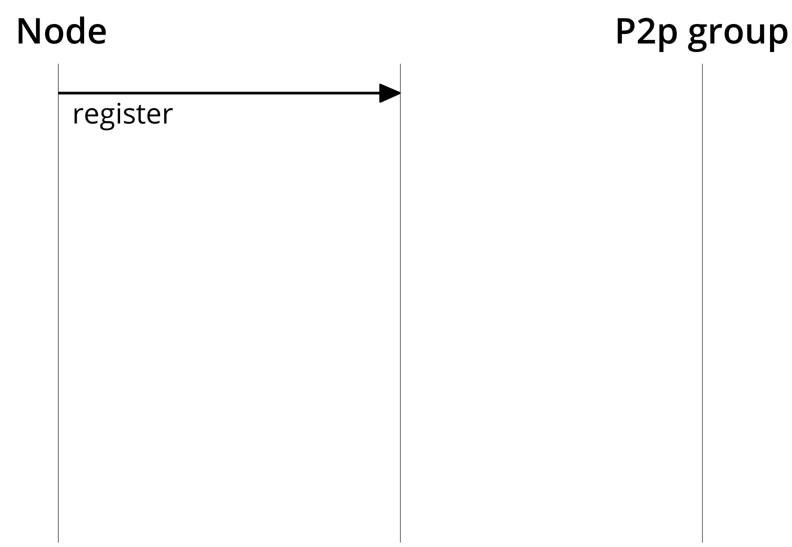
Then the member who receives the query will broadcast it to all other members of the group, who then will respond by sending their responses to the gossip-channel so that the query issuer does not get overwhelmed.



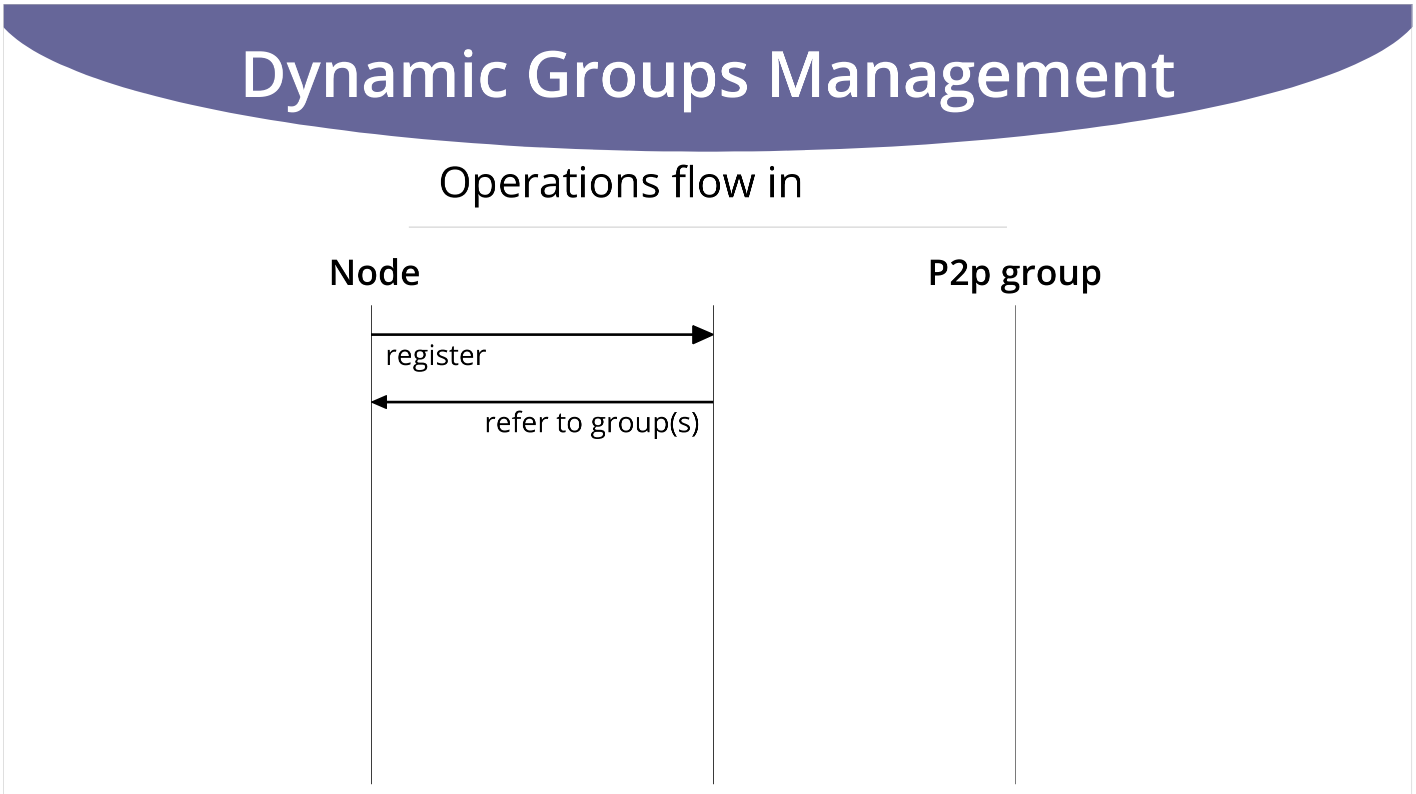
Ok, let me describe the end-to-end flow of operations in FOCUS so that it's clear how everything fits together.

Dynamic Groups Management

Operations flow in

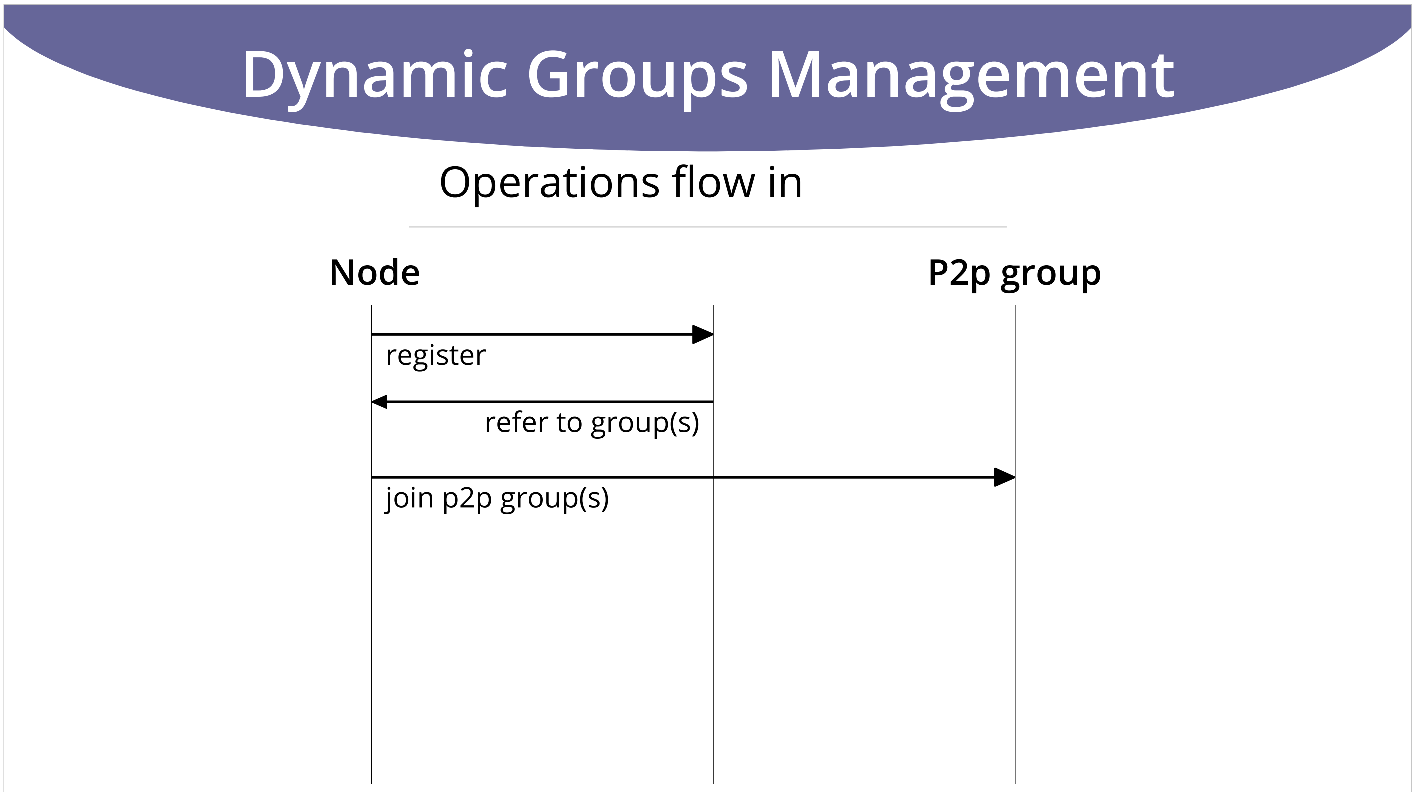


When a node starts up, its FOCUS agent will contact the FOCUS server and register its attributes.



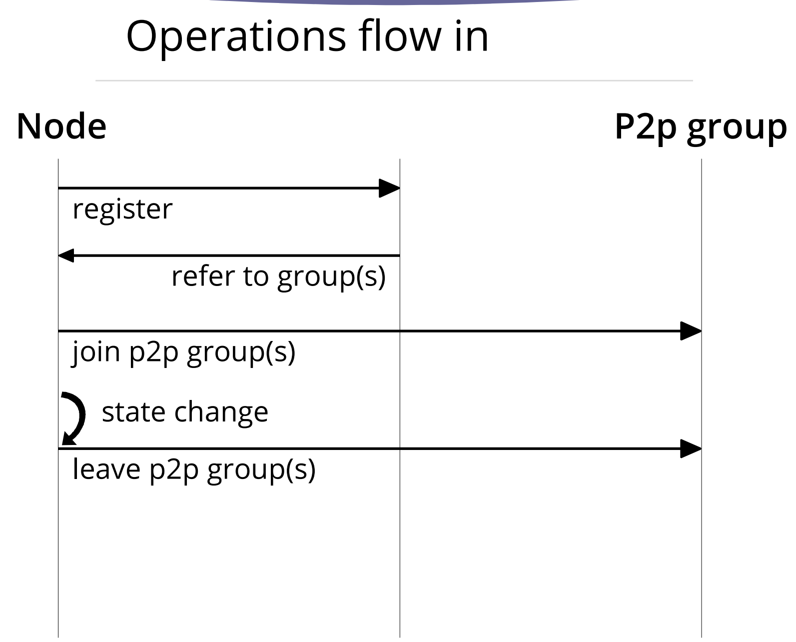
The FOCUS server will parse those attributes and look up the local information that it has about the already established p2p groups,

Then it will refer the newly registered node to the right groups.



Consequently, the node will go ahead and request to join each of those groups, and start gossiping with other members of the same group.

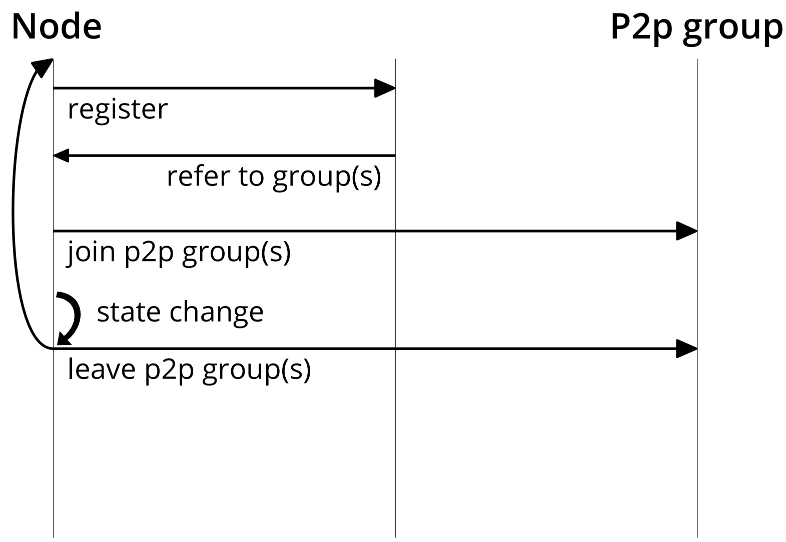
Dynamic Groups Management



Now, whenever one of the node’s attribute changes, the FOCUS node agent will detect that, and determine if we need to leave the corresponding group for that attribute (which is when the new value falls outside the group value range).

Dynamic Groups Management

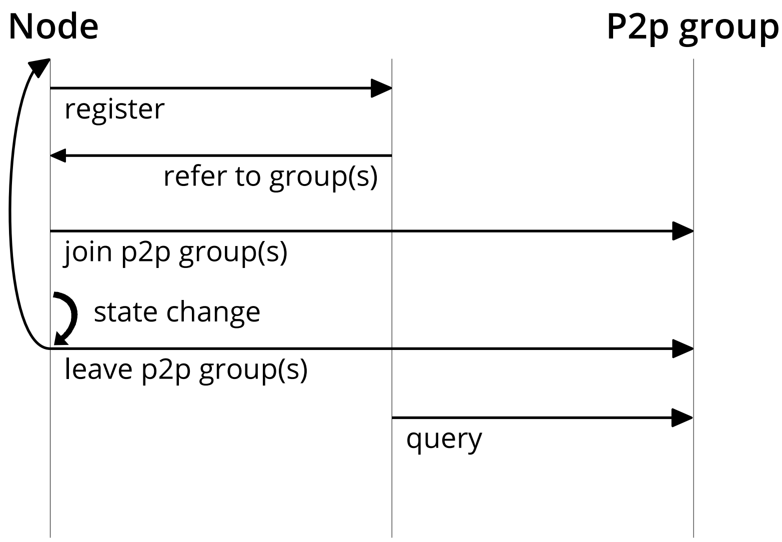
Operations flow in



If it decides to leave one of the groups, it will go back to and ask the FOCUS server for new group suggestions and so on.

Dynamic Groups Management

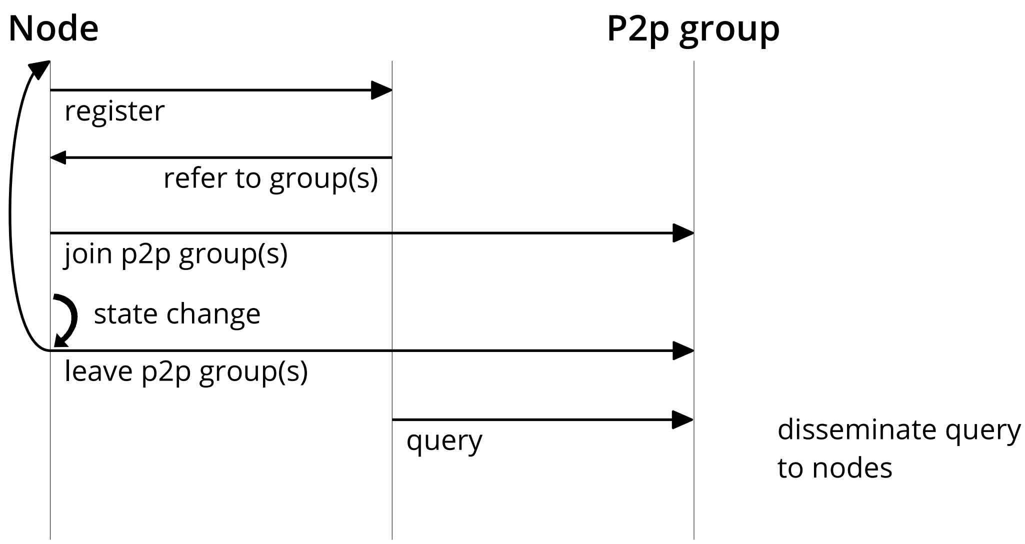
Operations flow in



Now, when the FOCUS server receives a query, it will decide which group to forward it to (based on the information it already has from the group representatives).

Dynamic Groups Management

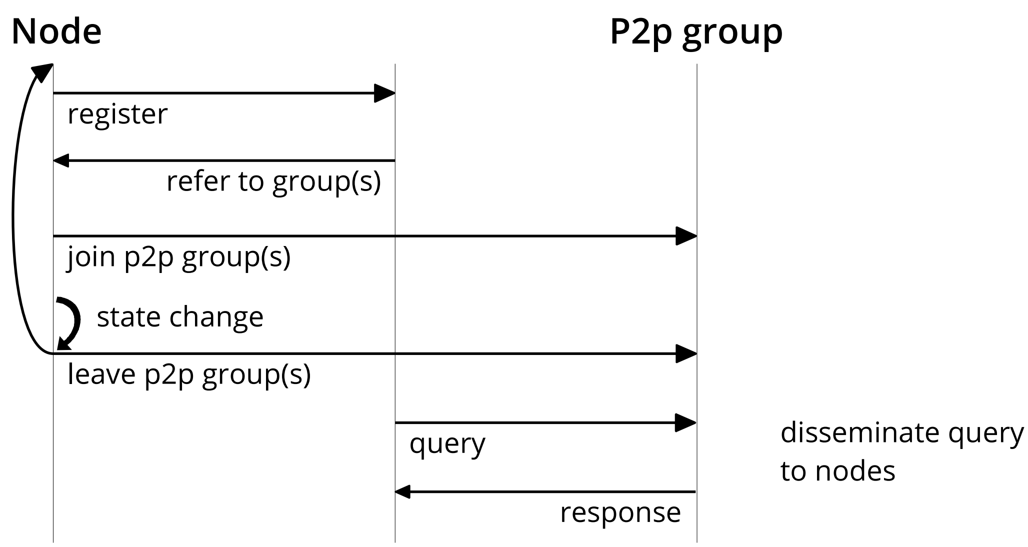
Operations flow in



Upon receiving a query from the FOCUS server, a member will disseminate it to other members of the group.

Dynamic Groups Management

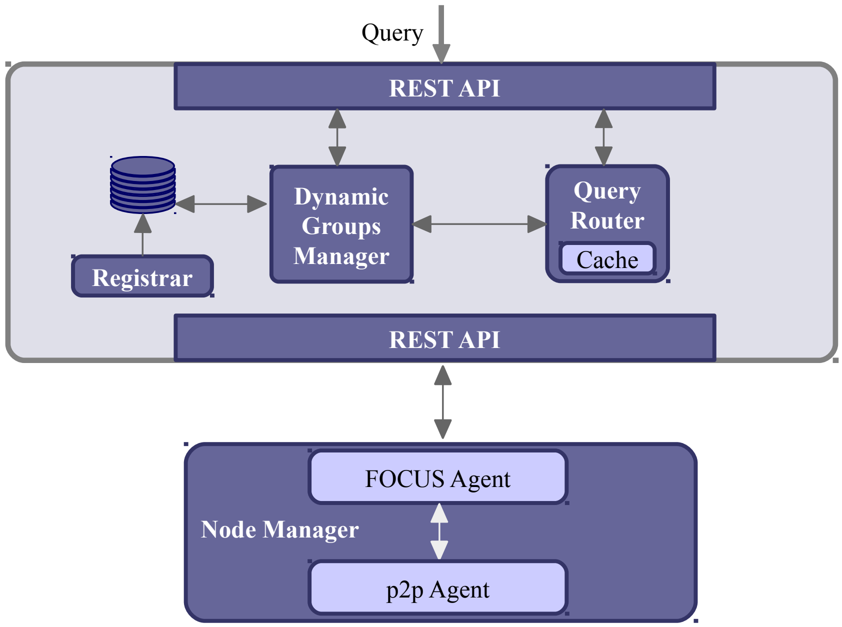
Operations flow in



The results then are aggregated and sent back to the FOCUS server.

So, now let's take a look at what tools we used to implement FOCUS, and how it performs.

Implementation



This is a detailed design of FOCUS, showing its internal components.

.

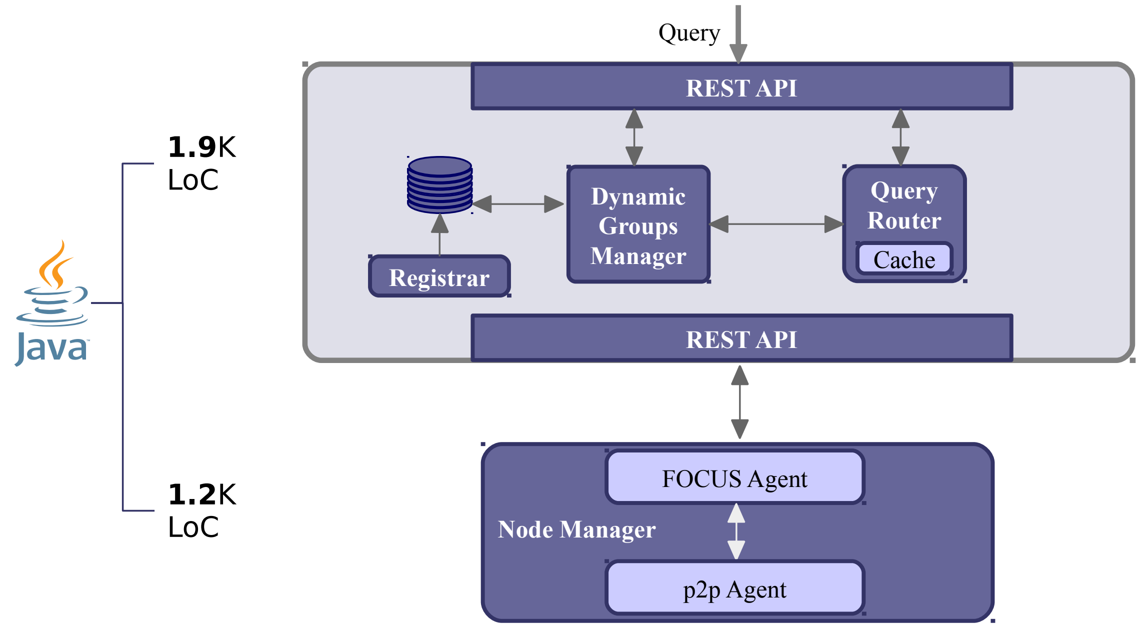
We have 3 main components, in addition to the node agent.

1- The Registrar, which is basically an entry point to the system (and holds static info).

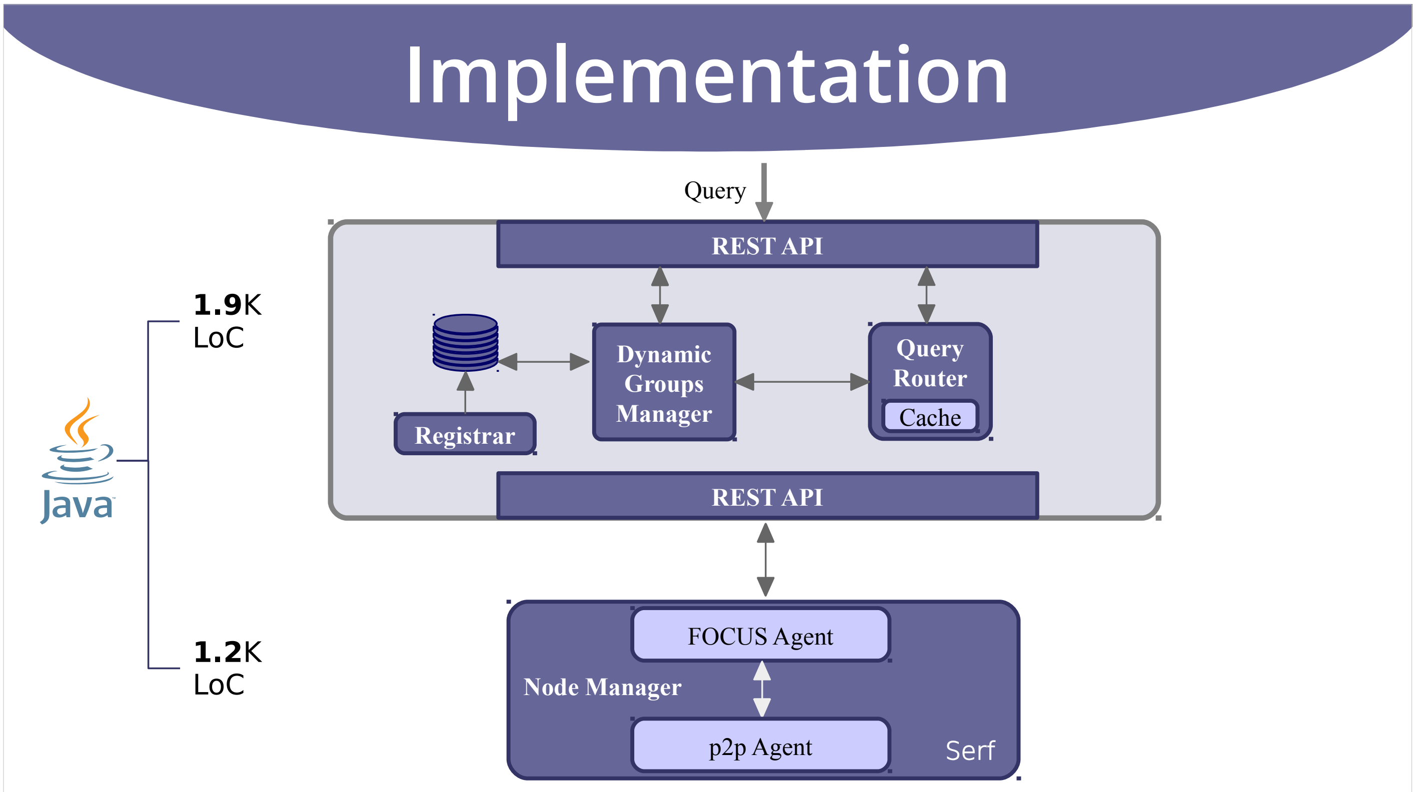
2- The Dynamic Groups Manager (DGM), which holds dynamic information on which nodes belong to which groups.

3- The Query Router: provides a REST API to FOCUS client applications (like OpenStack) to make queries and receive responses.

Implementation



FOCUS is implemented in Java with a total of 3.1K LoCs.



And here are some of the tools that we have used to implement FOCUS.

.

CLICK

SKIP

.

For our REST APIs, we used Eclipse's Jetty web server wrapper.

.

And for our data-store (which has mostly static information), we used the Apache Cassandra: a fast geo-distributed data store.

.

And we used HashiCorp's Serf as our gossip p2p fabric.

Evaluation

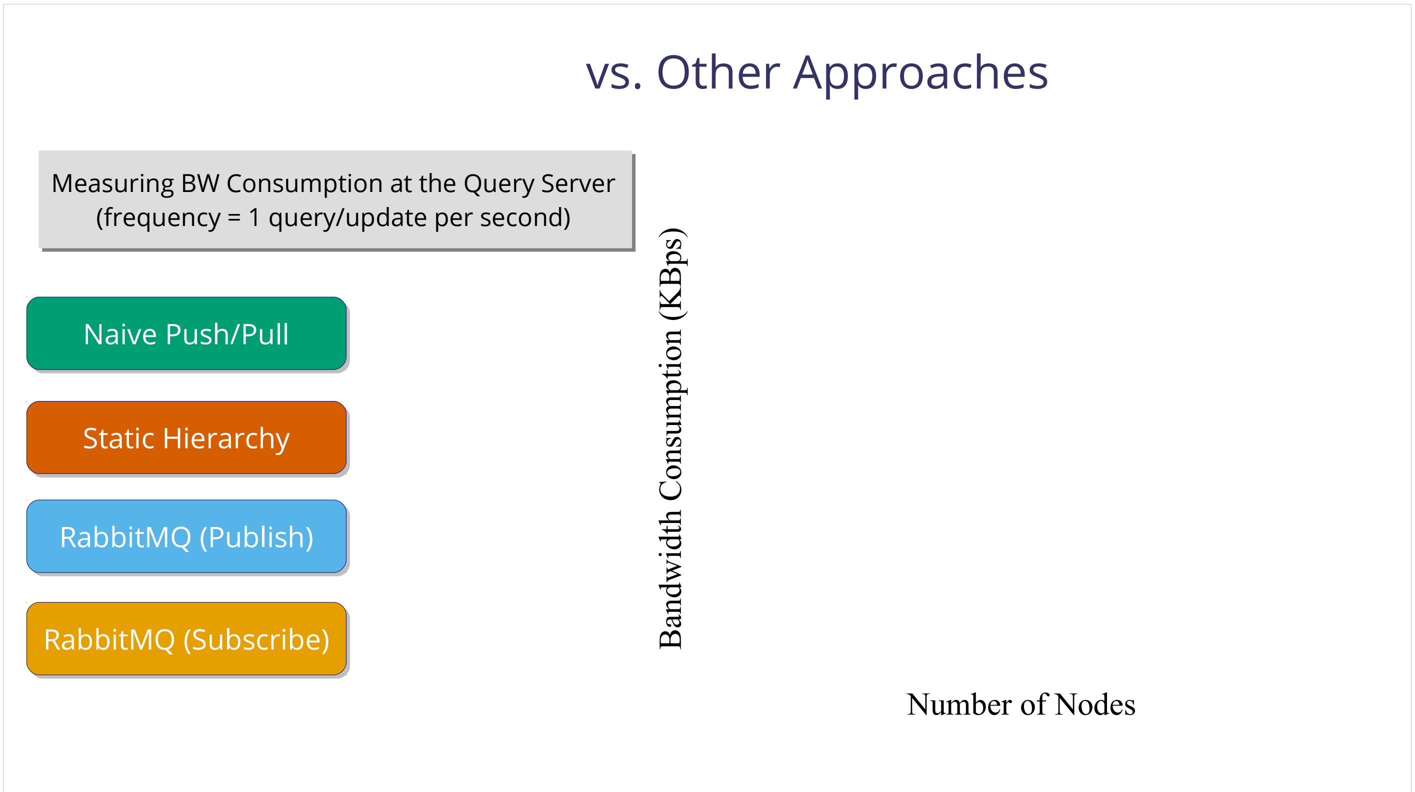
- Deployed in Amazon EC2
- 4 regions: Canada, California, Ohio, Oregon
- In each region: 8 VMs (4 vCPUs, 16GB RAM)
- FOCUS server running in California (same VM config)
- Testing up to 1600 simulated node agents

To evaluate FOCUS, we deployed it in 4 regions of Amazon's EC2.

.

And we ran tests with up to 1600 nodes.

Note that these are simulated nodes and we consolidated many simulated node agents in each of VMs we had.



In our first experiment, we compare FOCUS to other approaches of node finding.

.

We measured BW consumption on the query server and used that as an indicator for how well each of these approaches scale.

.

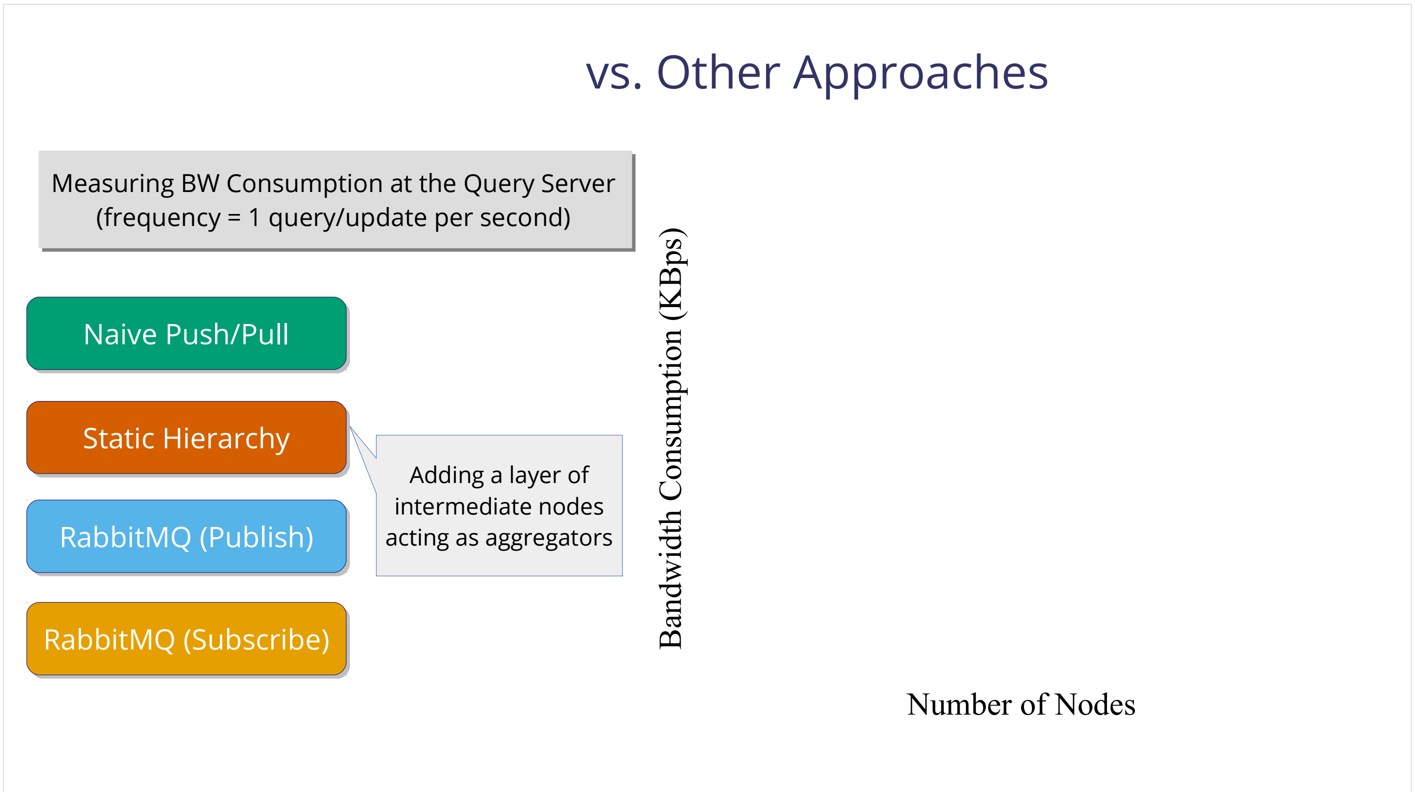
The query or update frequency is set to once per second.

…….

Let me describe really quickly what each of these approaches do.

.

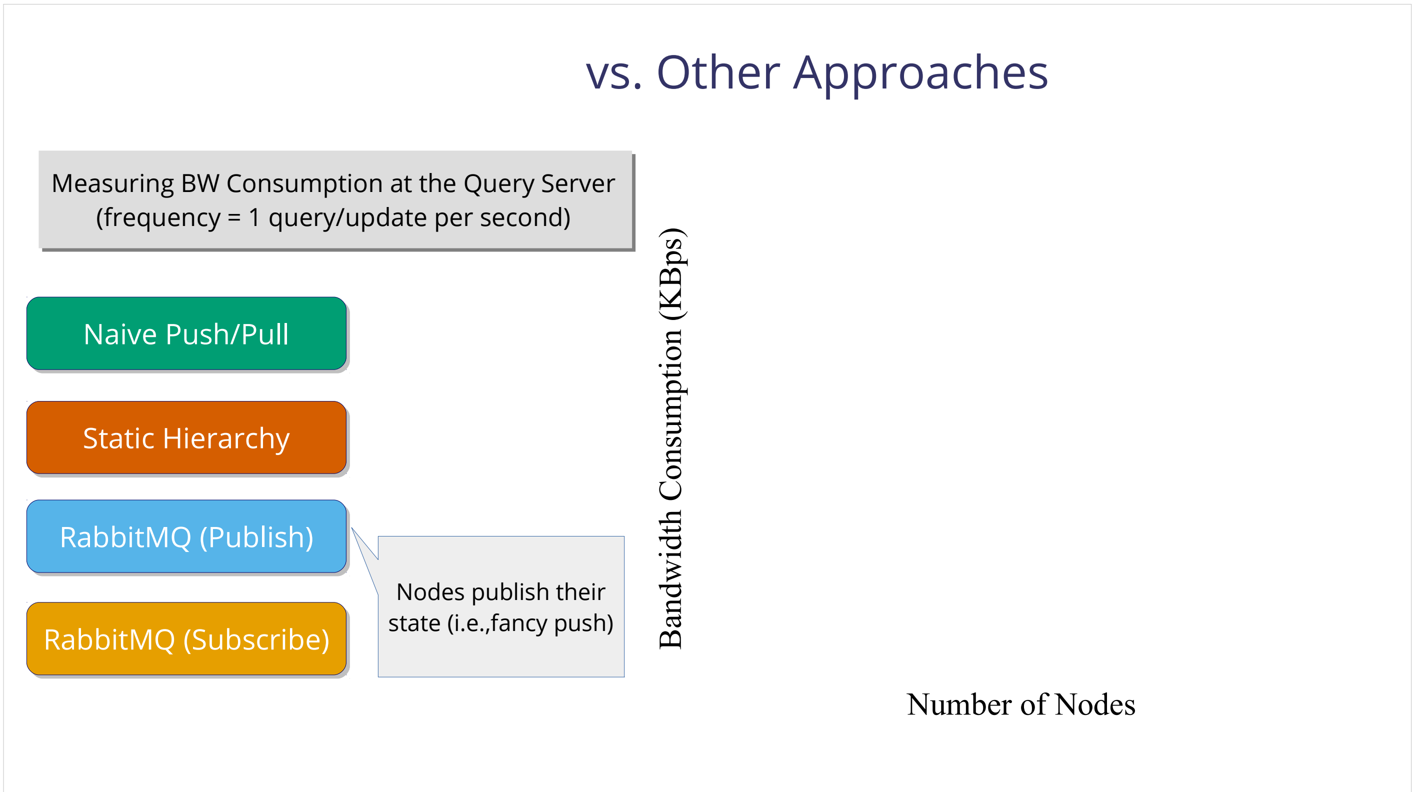
For the naive push/pull, we simply had nodes either push their information individually, or we made the FOCUS server pull from each of them.



We also implemented a system, in which we introduce static hierarchy.

.

In this system, we had a layer of intermediate nodes to which other nodes would send their updates. Who, in turn, will aggregate those updates and send them to the query server.



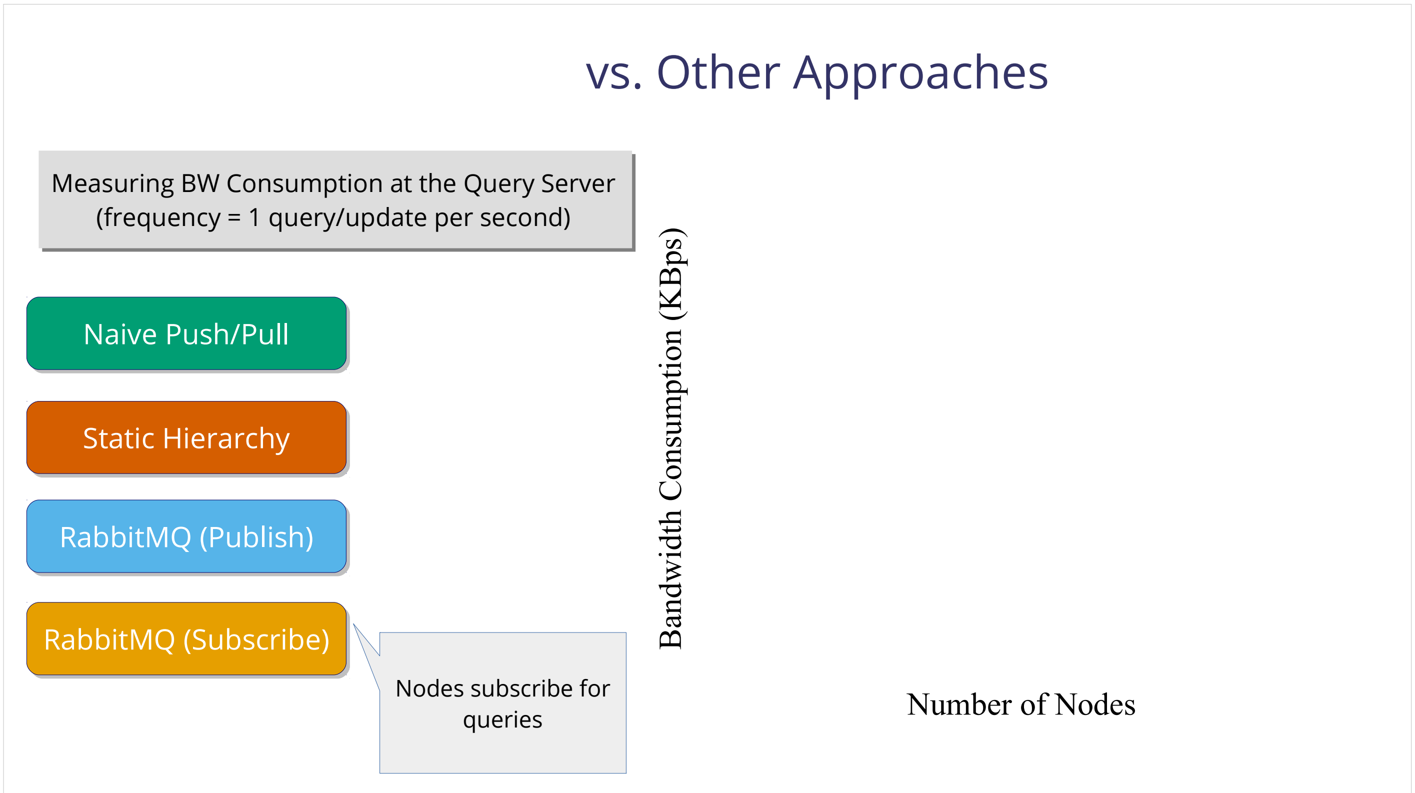
We also compare against RabbitMQ, the message Q used by OpenStack.

.

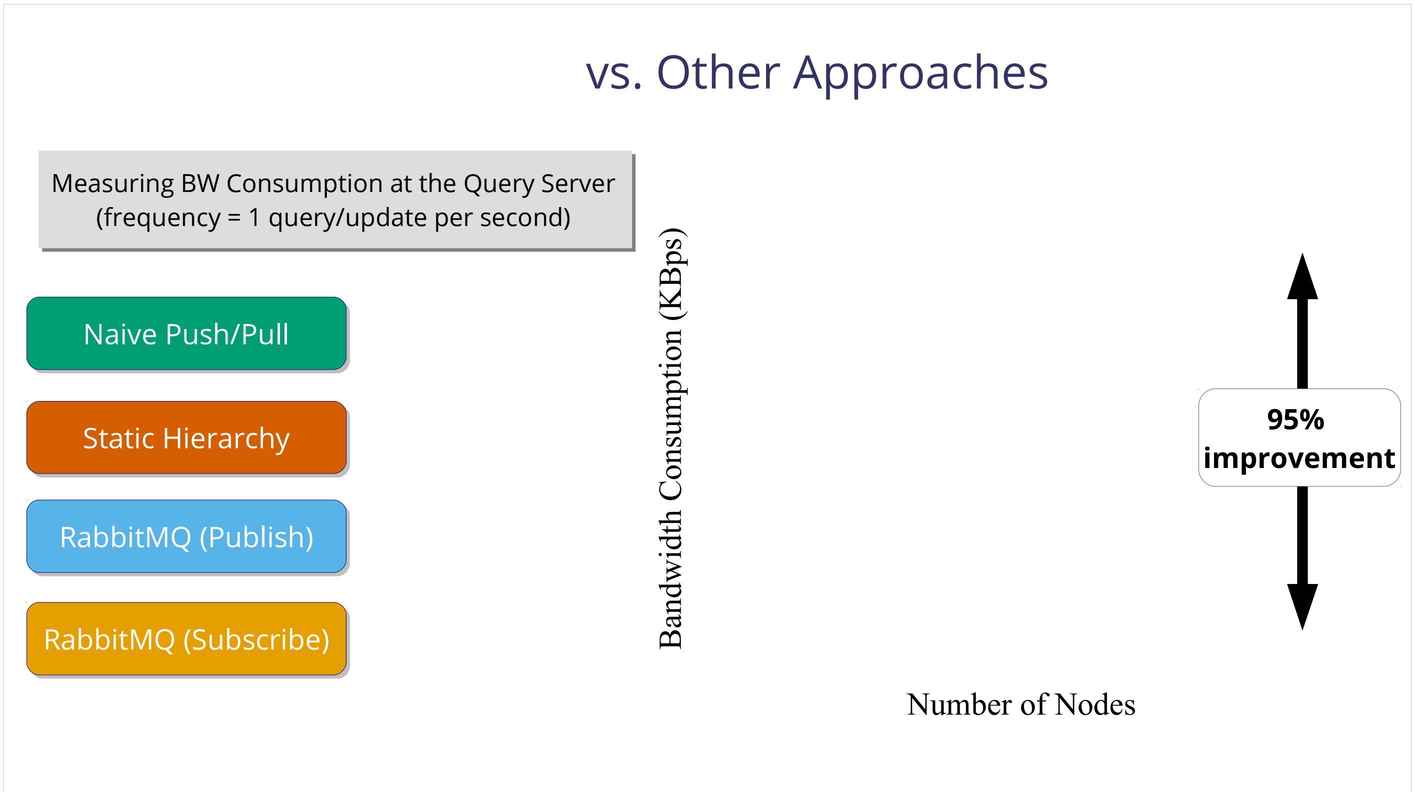
There are two configurations for RabbitMQ.

.

The publish configuration is where nodes publish their state (which is sort of a fancy push). And the query server will subscribe for those updates.



And the subscribe configuration is where nodes subscribe for queries, which are published by the Query Server.



Thanks to our loosely-coupled node management and attribute-based grouping, we were able to provide up to 95% improvement (or reduction of BW consumption) when compared with RabbitMQ subscribe configuration.

with Real-world Cloud Traces*

* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

In the next experiment, we evaluate FOCUS when processing real-world cloud queries.

with Real-world Cloud Traces*

75K OpenStack VM placement requests

* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

We had a trace of 75K OpenStack VM placement requests.

with Real-world Cloud Traces*

75K OpenStack VM placement requests

Replayed at accelerated rate (15,000x)

* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

And replayed the trace at an accelerated rate to test how FOCUS performed under high load.

with Real-world Cloud Traces*

75K OpenStack VM placement requests

Replayed at accelerated rate (15,000x)

Latency stabilizes after 600 nodes
→ because group size is capped (~150 nodes per group)

* “Chameleon Cloud: A configurable experimental environment for largescale cloud research,” https://www.chameleoncloud.org/.

You can see here that the latency stabilizes after having 600 nodes in the system.

.

This is due to the fact that we capped the group size (and hence, quickly respond to queries without compromising on accuracy).

Microbenchmarks

Resource usage of the FOCUS server (40 queries/s)

In this last part, we provide some micro-benchmarks for our FOCUS server as well as the node agent.

.

In this graph, we measured resource usage of the FOCUS server under a heavy load, where it processed 40 queries / second.

.

And thanks to the loosely-coupled design, you can see that it's not resource hungry.

Microbenchmarks

Resource usage of the FOCUS server (40 queries/s)

Overhead imposed by node agent (KBps)

We have also measured the overhead imposed by our node agent (especially for the p2p gossip-part)

.

We had two configurations for this experiment:

1- when the node processes 1 Query / second

2- and the normal behavior of a node (i.e., when it gossips with other nodes.)

.

The overhead here is negligible even if we have large groups of up to 500 nodes.

.

The reason here is because in normal operation, the number of peer nodes with which we gossip is irrelevant to the group size.

Microbenchmarks

Resource usage of the FOCUS server (40 queries/s)	Overhead imposed by node agent (KBps)	Query response time for different group sizes

In this last experiment, we measure the query response latency when we vary the group size (which has a direct impact on how quickly the group converges).

.

In addition, we measure the latency if the query response is to be fetched from a local cache.

.

You can see that query response latency does not grow rapidly as the group size increases.

Conclusion

- **Current systems' scalability is limited**
  - This is due to tightly-coupled node management

To conclude with,
We showed that current systems' scalability is limited.
.
We studied the underlying cause of this and concluded that the reason of this limitation is because such systems impose an unneeded tightly-coupled node management.

Conclusion

- **Current systems' scalability is limited**
  - This is due to tightly-coupled node management
- **FOCUS is scalable search service**
  - Employs a *loosely-coupled* node management (p2p)
  - *Scales* better than current approaches (15x improvement)
  - Imposes *minimal* overhead on nodes
  - *Integrates* well with current systems

To that end, we designed FOCUS: a scalable search service for distributed systems.

.

Through loosely-coupled node management and attribute-based grouping, we were able to scale much better than current approaches.

And at the same time impose a minimal overhead on the nodes in the system.

.

We also believe in practicality, so we designed FOCUS in a way that it allows it to integrate easily well with existing systems.

Thank You!

Questions?