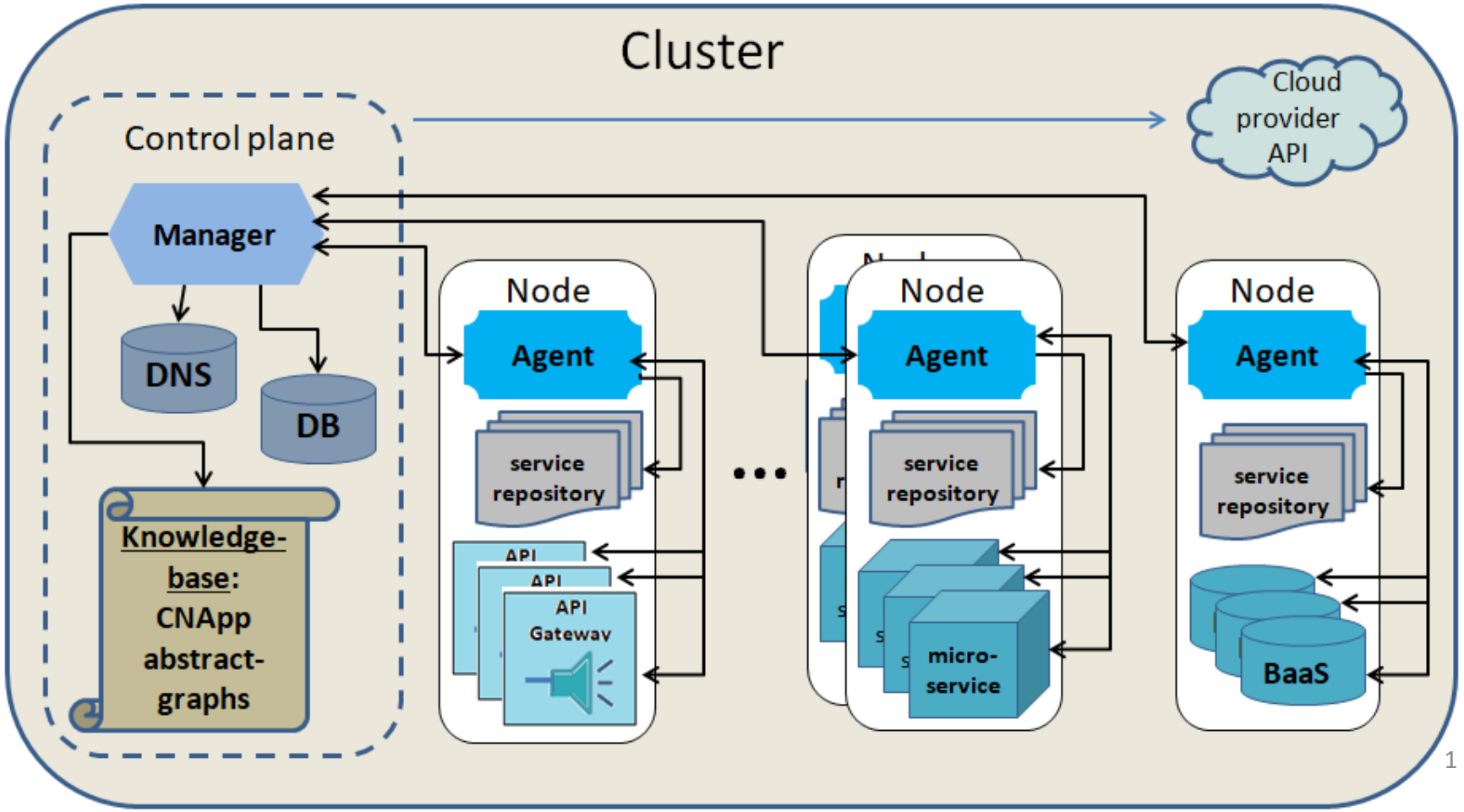
SSMMP: Simple Service Mesh Management Protocol

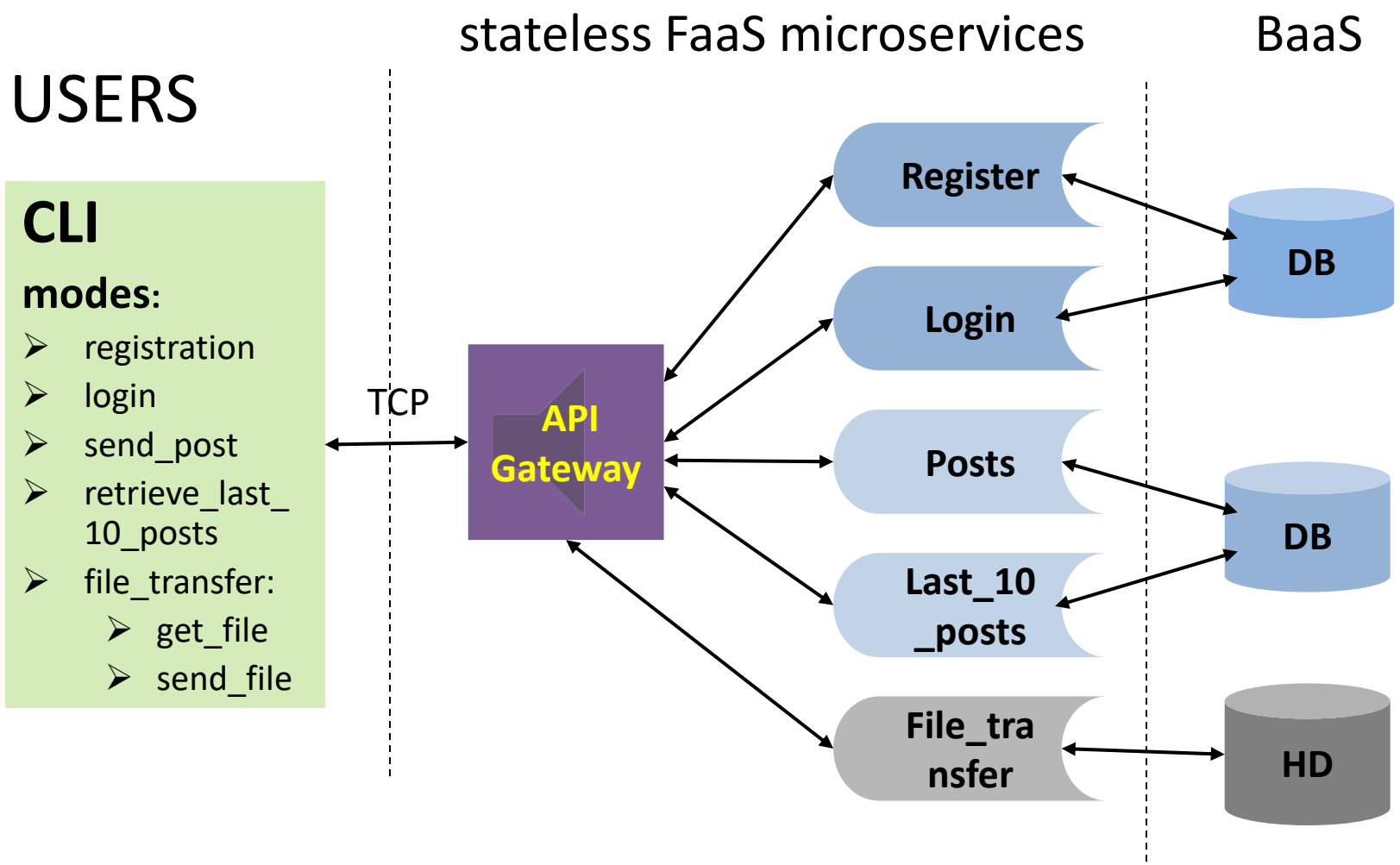


Abstract

- A simple protocol for Service Mesh management is proposed.
- The protocol specification consists of the formats of messages, and the actions taken by senders and recipients.

- The idea is that microservices of Cloud-Native Application should be also involved in configurations of their communication sessions. It does not interfere with the business logic of the microservices and requires only minor and generic modifications of the microservices codebase, limited only to network connections.
- The proposed protocol has been implemented (as a proof of concept) for simple social media CNApps.
- The implementations clearly show that SSMMP should be viewed (by developers) as an integral part of CNApps.

A simple test CNApp for the SSMMP protocol



Simple CNApp

- The basic functionality of the application is as follows.

  User interface is Command Line Interface (CLI).

  CLI communicates (via TCP) only with API Gateway.

  The API Gateway forwards user requests to appropriate microservices: registration, login, table (retrieving the recent 10 post of the users), storing user posts in a DB, and File transfer to store (upload) and retrieve (download) files to/from HD.

  Responses from microservices are sent back to API Gateway, and then forwarded to the users.

  Microservices can not communicate directly with users; only via the API Gateway.

  API Gateway and microservices are stateless.

  Requests and responses are objects of class String.

Simple CNApp: all communication is based on predefined protocols

All messages (requests and responses) are 7 bit ASCII code strings

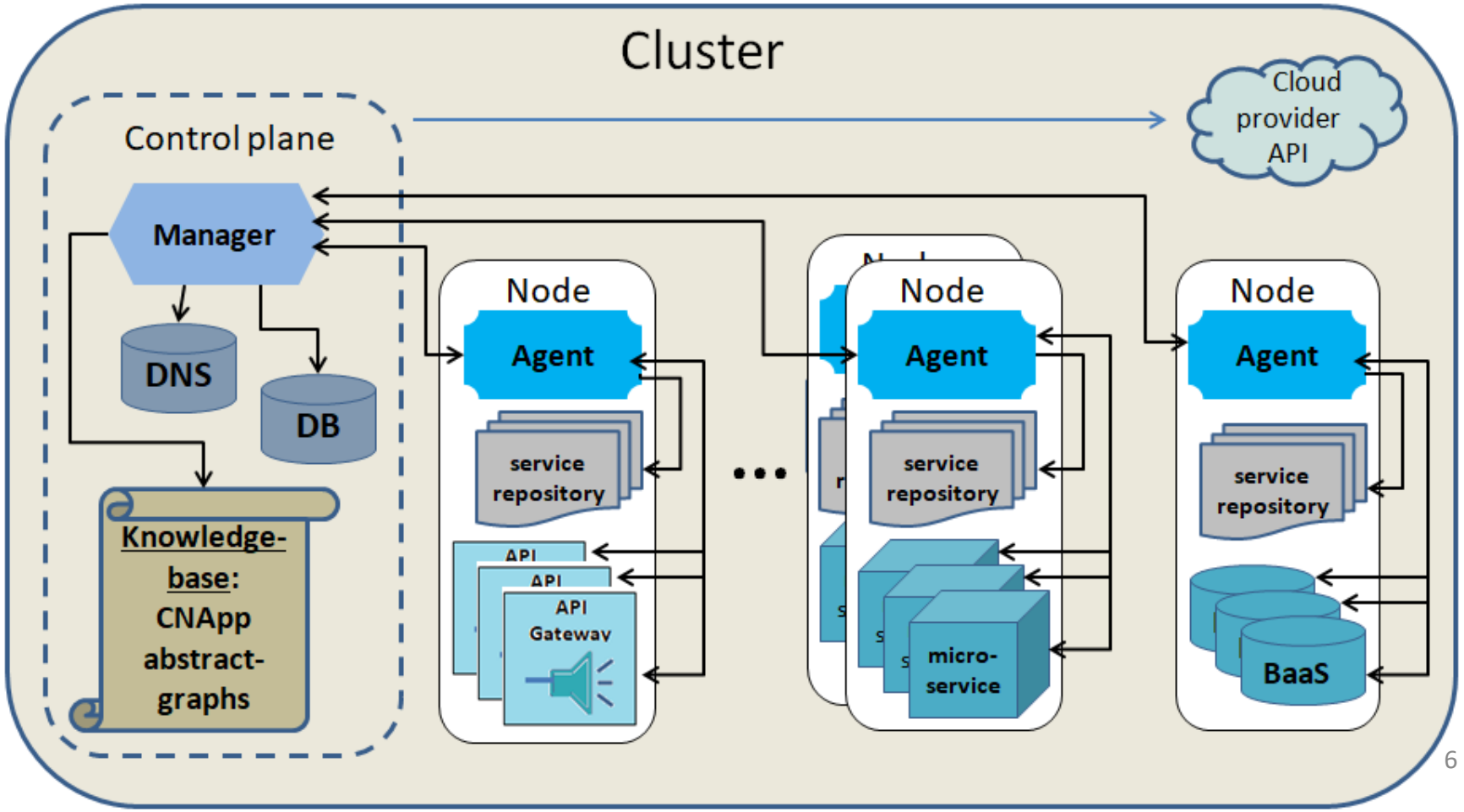
- Format of a request

```
Type: registration_request
Message_id: 30970
Line_1:
Line_2:
…
```

- Format of the response

```
Type: registration_response
Message_id: 30970            //the same as in the request
Status: 200
Line_3:
Line_4:
…
```

SSMMP: a simple protocol for Service Mesh management



CNApp

- Cloud Native Application (CNApp) is a distributed application composed of microservices and deployed in the Cloud.
- Microservices constitute an architectural pattern where a complex and sophisticated application is made up of a collection of fine-grained, self-contained microservices that are developed and deployed independently of each other.

- Netflix uses over 1000 microservices now. Uber now has 4000 or more independent microservices.
- There are also Amazon, Facebook, Google, and Spotify Google, ChatGTP, 5G infrastructures etc

- **A distributed control plane is necessary for such huge CNApps!**

Service Mesh

- Service Mesh is an infrastructure for CNApps that allows to transparently add security, observability and management of network traffic between the microservices (?without interfering with the codebase of the microservices?).

- Usually, Service Mesh is built on the top of Kubernetes and Docker.

- Each microservice is equipped with its own local proxy (called sidecar). Sidecars can be automatically injected into Kubernetes pods, and can transparently capture all microservice traffic.

- The sidecars form the data plane of Service Mesh.

Service Mesh

- The control plane of Service Mesh is (logically) one manager responsible for configuring all proxies in the data plane to route traffic between microservices and load balancing, and to provide resiliency and security.

- **Linkerd** and **Istio** both extending Kubernetes, are the best known and most popular open source software platforms for realizing Service Mesh.

- Istio uses Envoy's proxy while Linkerd uses its own specialized micro-proxies.

- **Cilium** is a new, interesting and also open source software platform

SSMMP

- How to automate the executing, scaling and reconfiguration of Cloud-Native Apps in a general way, but not at the software level?

- Following Mulligan 2023 this automation can be accomplished by implementing a generic protocol that extends the networking stack, on the top of TCP/IP.

- The solution we propose is the Simple Service Mesh Management Protocol (SSMMP) as a specification to be implemented in a Cloud cluster.

SSMMP

- The specification consists of the formats of messages exchanged between the parties (actors) to the conversation of the protocol, and the actions taken by the senders and receivers of the messages.

- The actors are: Manager, agents (residing on the nodes that make up the cluster), and instances of microservices running on these nodes.

- Similar to Kubernetes clusters.
- The main difference is the abstract architecture of CNApps and simple general rules for the automation of CNApps management.

Microservices

- Microservice architecture comprises services that are fine-grained and protocols that are lightweight.

- CNApp is a network application where microservices communicate with each other by exchanging messages (following CNApp's business logic) using dedicated, specific protocols implemented on top of the network protocol stack. Usually, it is TCP/UDP/IP.

- Each of these protocols is based on the client-server model of communication.

Microservices

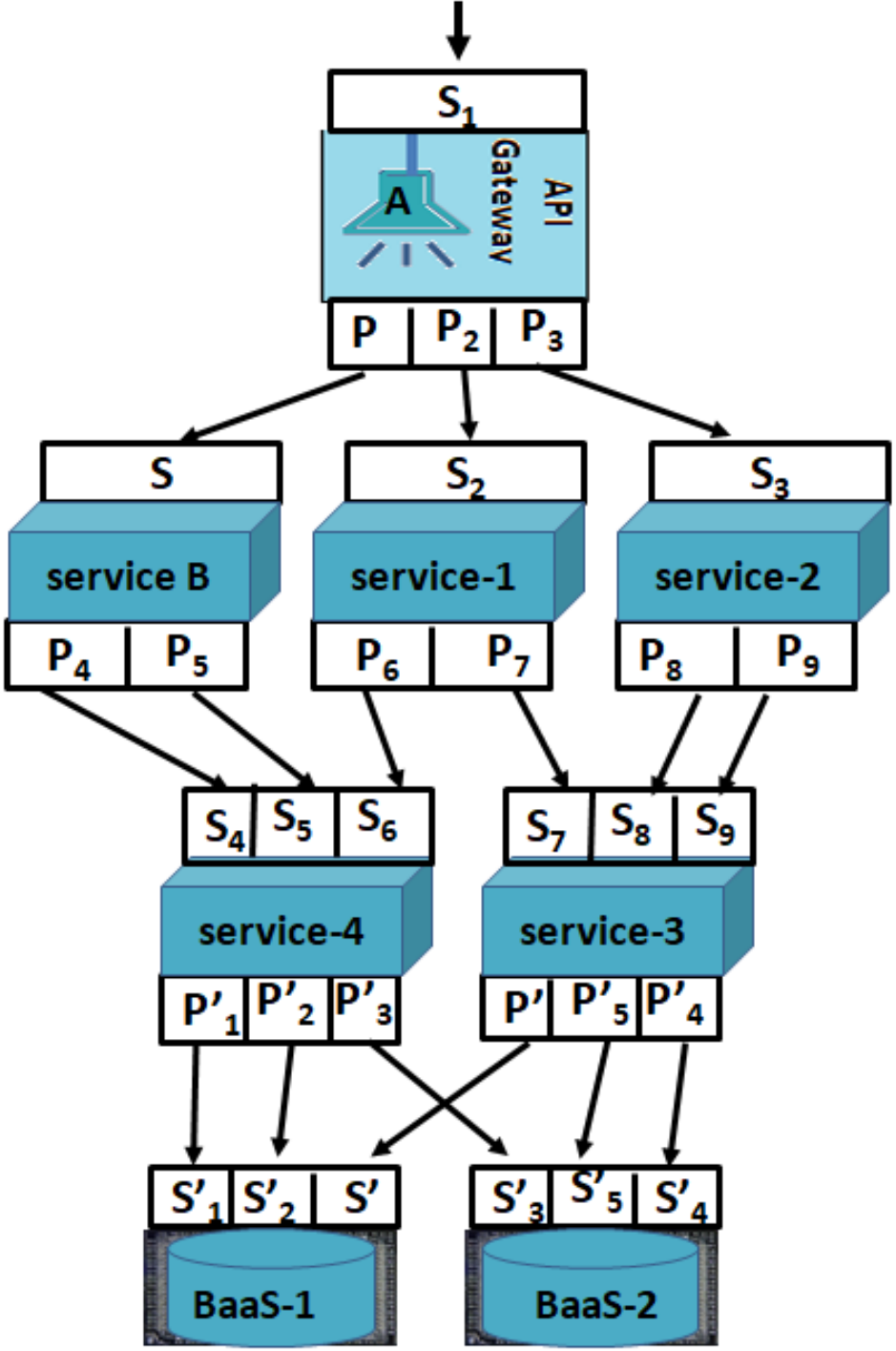
- The server (as part of a running microservice on a host with a network address) is listening on a fixed port for a client that is a part of another microservice, usually running on a different host.

- Since a client initiates a communication session with the server, this client must know the address and port number of the server.

- A single microservice can implement and participate in many different protocols, acting as a client and/or as a server.

- Thus, a microservice can be roughly defined as a collection of servers and clients of the protocols it participates in, and its own internal functionality (business logic)

Simple example of an abstract CNApp



Communication protocols

- Communication protocols (at application layer) are defined as more or less formal specifications independently of their implementation.
- Protocol is denoted (P,S) as two closely related parties to the conversation: the server S and the client P which are to be implemented on two microservices.
- After implementation, they are integral parts (modules) of microservices that communicate using this protocol.

- Abstract inputs of a microservice can be defined as a collection of the servers (of the protocols) it implements:

$$\mathbf{IN := (S_1, S_2, \dots , S_k)}$$

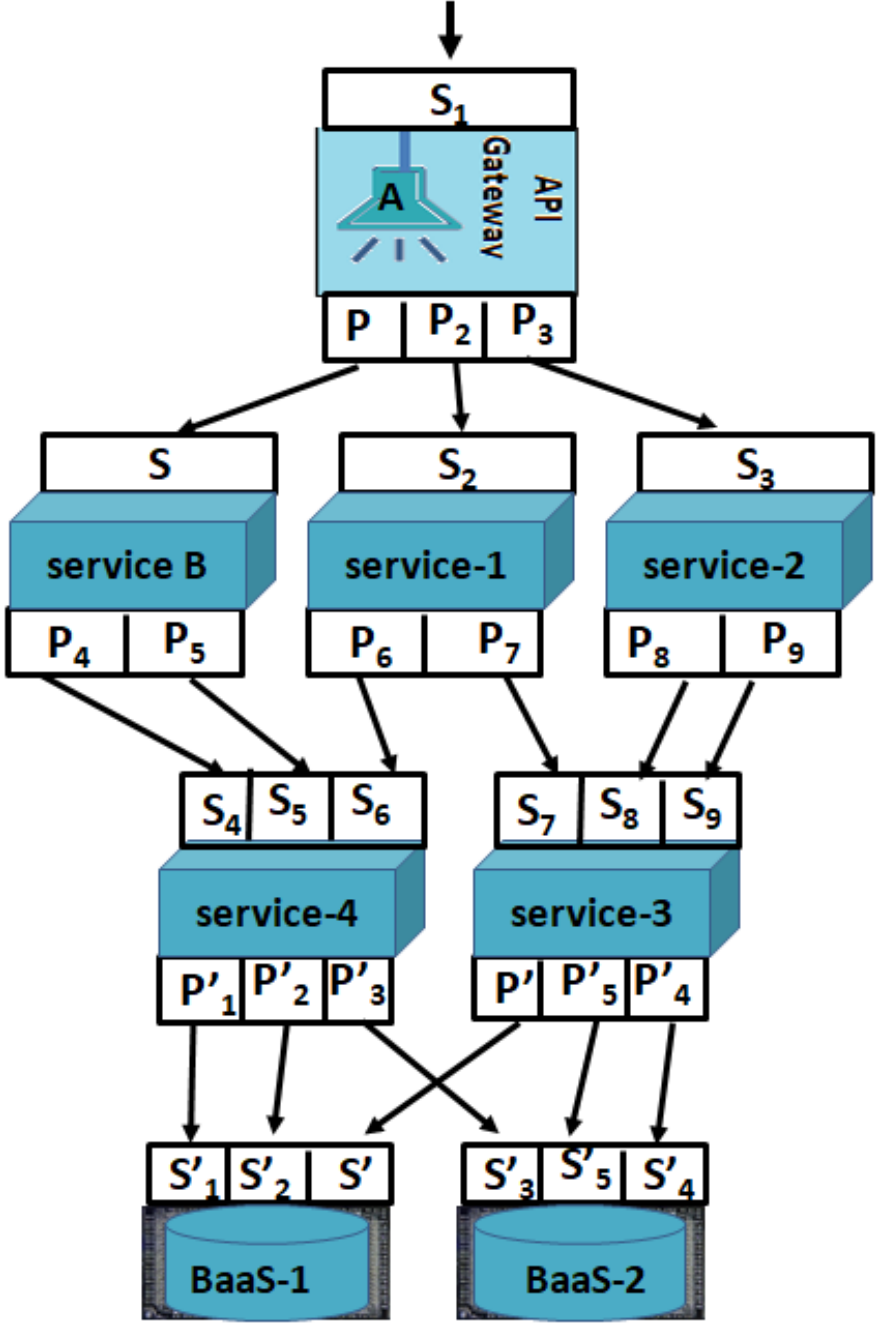
Communication protocols

- Abstract outputs of a microservice is defined as a collection of the clients (of the protocols) it implements:

$$\mathbf{OUT := (P'_1, P'_2, \ldots, P'_n)}$$

- Components of abstract input are called *abstract sockets*, whereas components of abstract output are called *abstract plugs*.

- An abstract plug (of one microservice) can be associated to an abstract socket (of another microservice) if they are two complementary parties of the same communication protocol.



On the left, a directed acyclic graph represents a workflow of microservies that comprise a simple CNApp.

The edges of the graph are of the form (abstract plug -> abstract socket).

They are directed, which means that a client (of a protocol) can initiate a communication session with a server of the same protocol.

Abstract definition of microservice

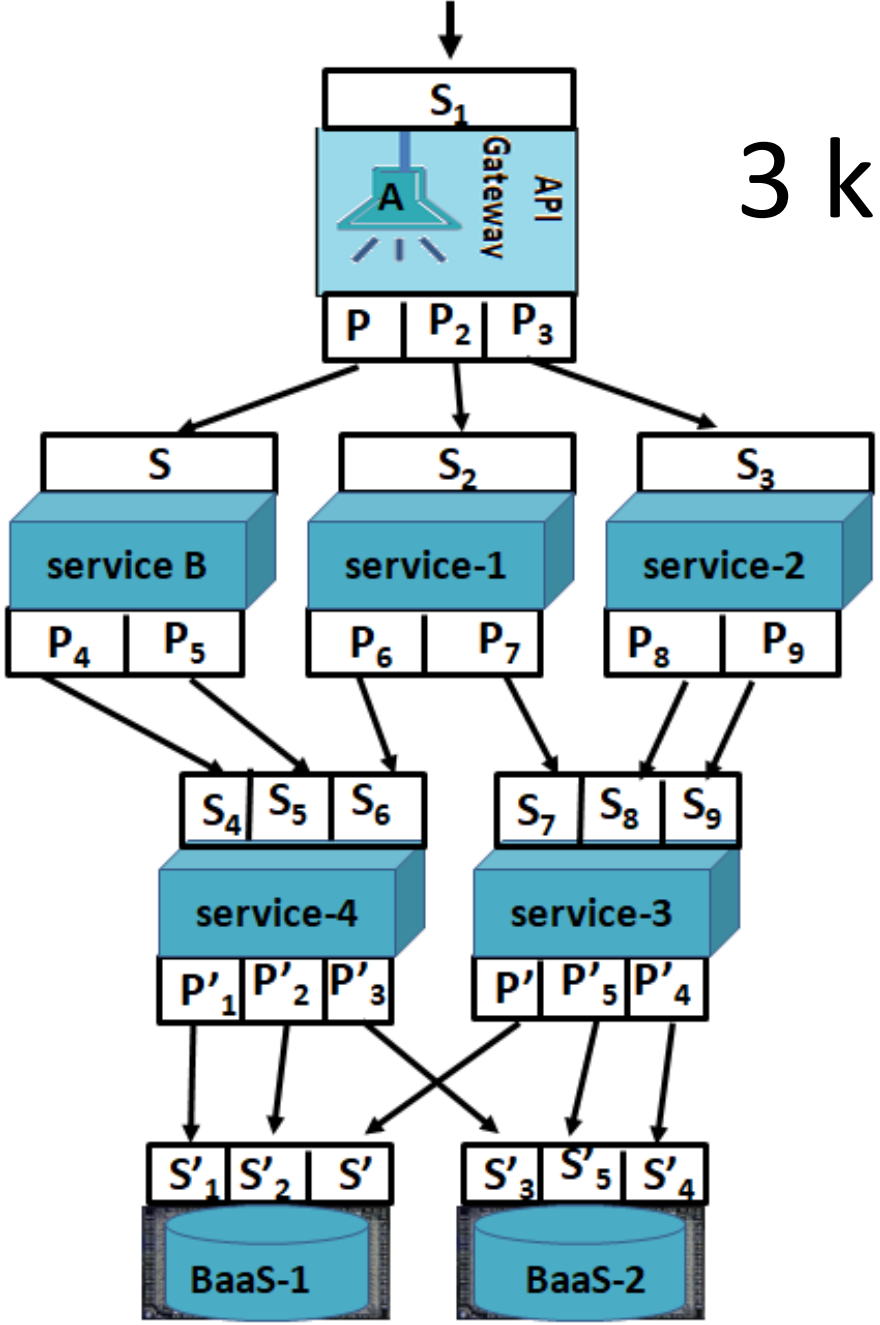
- *Microservice* is defined as

A:= (IN. F, OUT))

- where **IN** is the abstract inputs of the microservice, **OUT** is the abstract outputs, and **F** denotes the business logic of the microservice.
- Incoming messages, via abstract sockets of **IN** or/and via abstract plugs of **OUT**, invoke (as events) functions that comprise the internal functionality **F** of the microservice.
- This results in outgoing messages sent via **IN** or/and **OUT**.

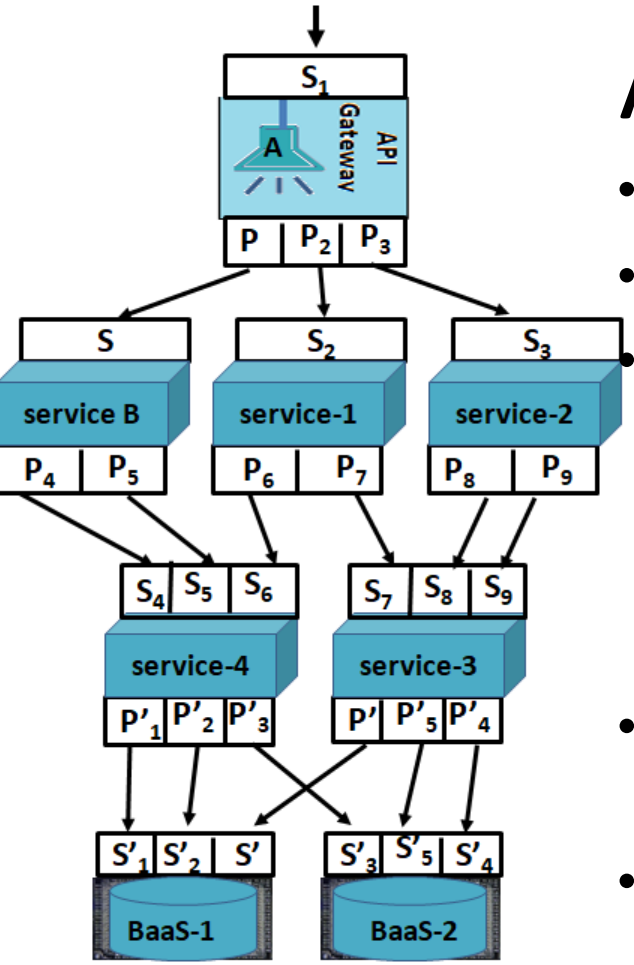
3 kinds of microservices

1. API Gateways - entry points of CNApp for users. Usually, IN of API Gateway has only one element. Its functionality comprises in forwarding users requests to appropriate microservices. API Gateway is stateless.
2. Regular microservices - their IN and OUT are not empty. They are also stateless. Persistent data (states) of these microservices are stored in backend storage services (BaaS).
3. Backend storage services (BaaS) where all data and files of CNApp are stored. Their OUT is empty.



3 kinds of microservices

- All of them are also called *services* of CNApp.
- Figure on the left illustrates a CNApp composed of one API Gateway, five stateless regular microservices, and two backend storage services (BaaS).
- The edges denote abstract connections and can also be seen as abstract compositions of services within a workflow.



Abstract graph of CNApp

- A directed labeled multi-graph: **G := (V, E)**
- **V** and **E** denote respectively Vertices and Edges.
- Vertices **V** is a collection of names of services of CNApp, i.e. elements denoted in Figure on the left as: A (the API Gateway); regular microservices: service B, service-1, service-2, service-3, and service-4; and BaaS services: BaaS-1 and BaaS-2.
- Edges **E** is a collection of labeled edges of the graph. Each edge is of the form: (C, (P,S), D)
- where C and D belong to **V** and (P,S) denotes a protocol.
- That is, P belongs to OUT of C, and S belongs to IN of D. The edges correspond to *abstract connections* between microservices.

Abstract graph of CNApp

- The direction of an edge in the graph represents the client-server order of establishing a concrete connection.

- An implementation of abstract connection (C, (P,S), D) in a running CNApp results in a concrete plug (in an instance of service C) corresponding to this abstract plug P. The concrete plug is connected to a concrete socket (corresponding to abstract socket S) of an instance of service D.

- This connection is called a *communication session*

Abstract graph of CNApp

- Initial vertices of the abstract graph correspond to API Gateways (entry points for users), whereas the terminal vertices correspond to backend storage services (BaaS) where all data and files of the CNApp are stored.

- The vertices representing regular microservices are between the API Gateways and the backend storage services (BaaS).

- Scaling through replication and reduction (closing replicas) of a service forces it to be stateless.

- API Gateways and regular microservices are stateless and can be replicated, i.e. multiple instances of such a service can run simultaneously.

Abstract graph of CNApp

- To run CNApp, instances of its services must first be executed, then abstract connections can be configured and established as real connections, and finally protocol sessions (corresponding to these connections) can be started.

- Some services and/or connections may not be used by some executions of CNApp. Temporary protocol sessions can be started for already established connections (and then closed along with their connections) dynamically at runtime.

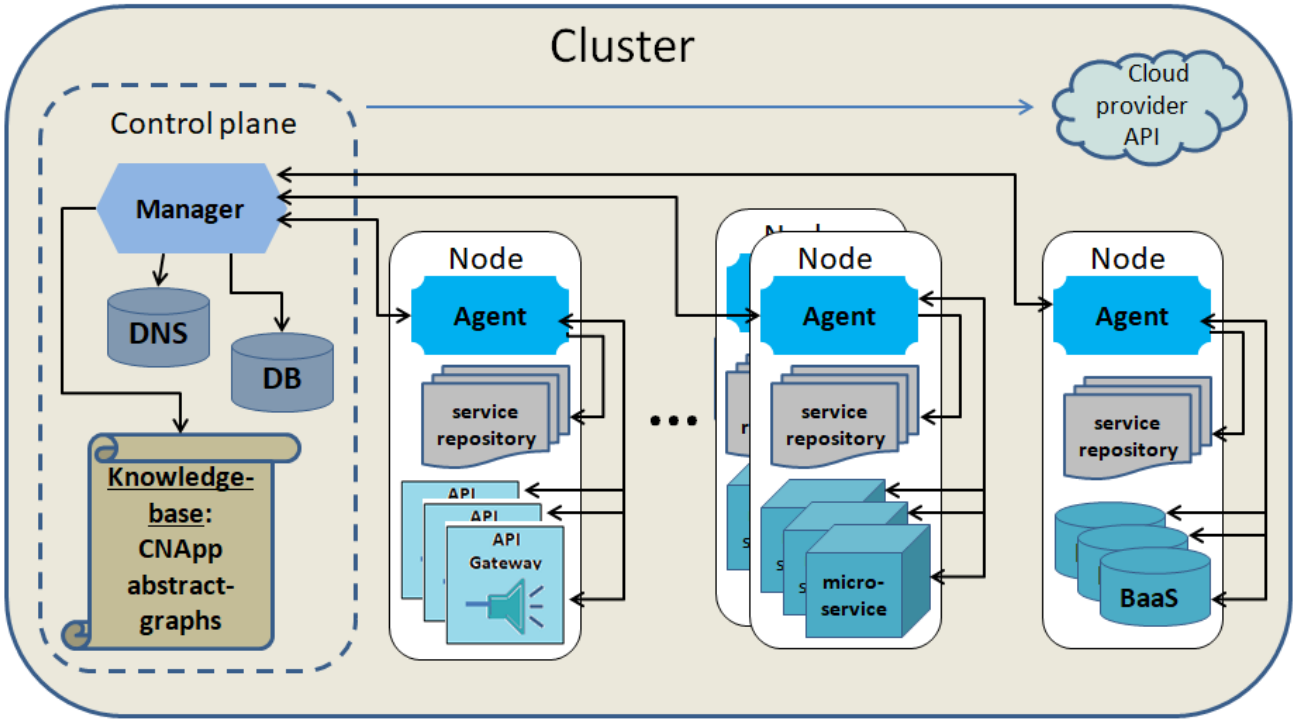
Abstract graph of CNApp

- Multiple service instances may be running, and some are shutting down.

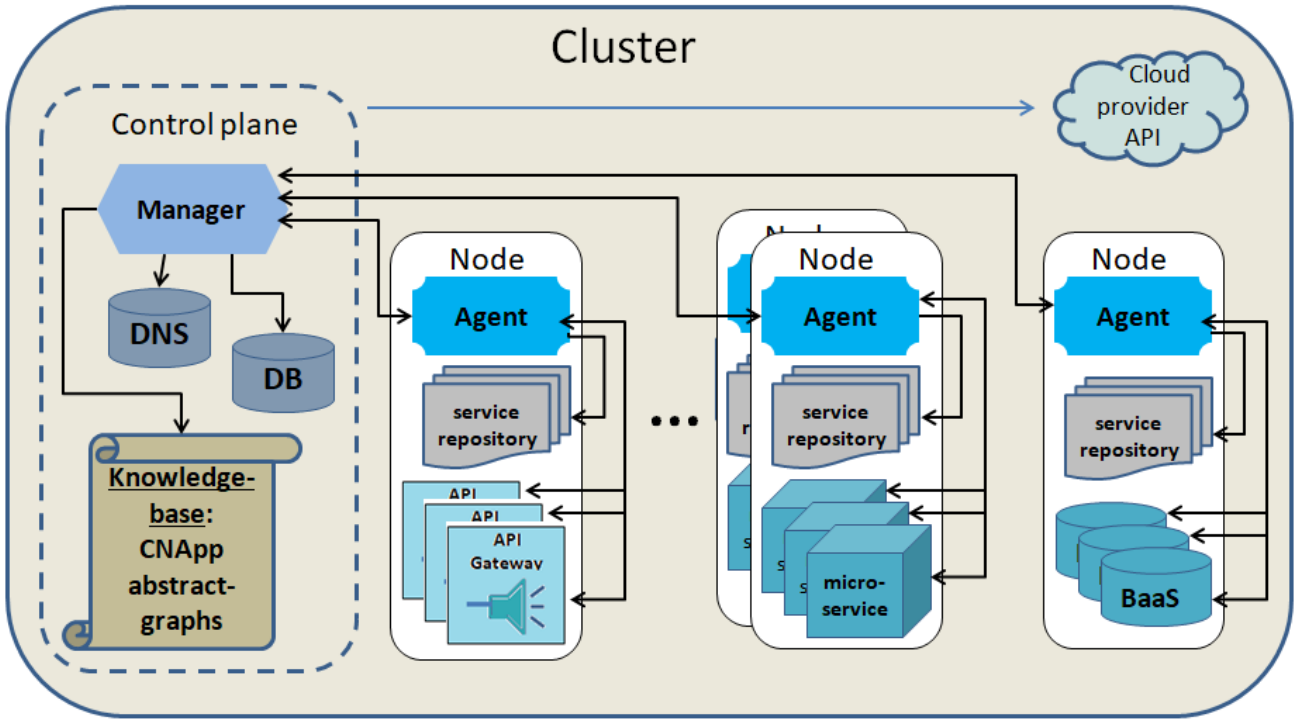
- This requires dynamic configurations of network addresses and port numbers for plugs and sockets of the instances.

- The novelty of SSMMP lies in the smart use of these configurations. A similar idea has been used by Netflix at the software level, but has not been fully explored.

Simple service mesh management protocol - SSMMP



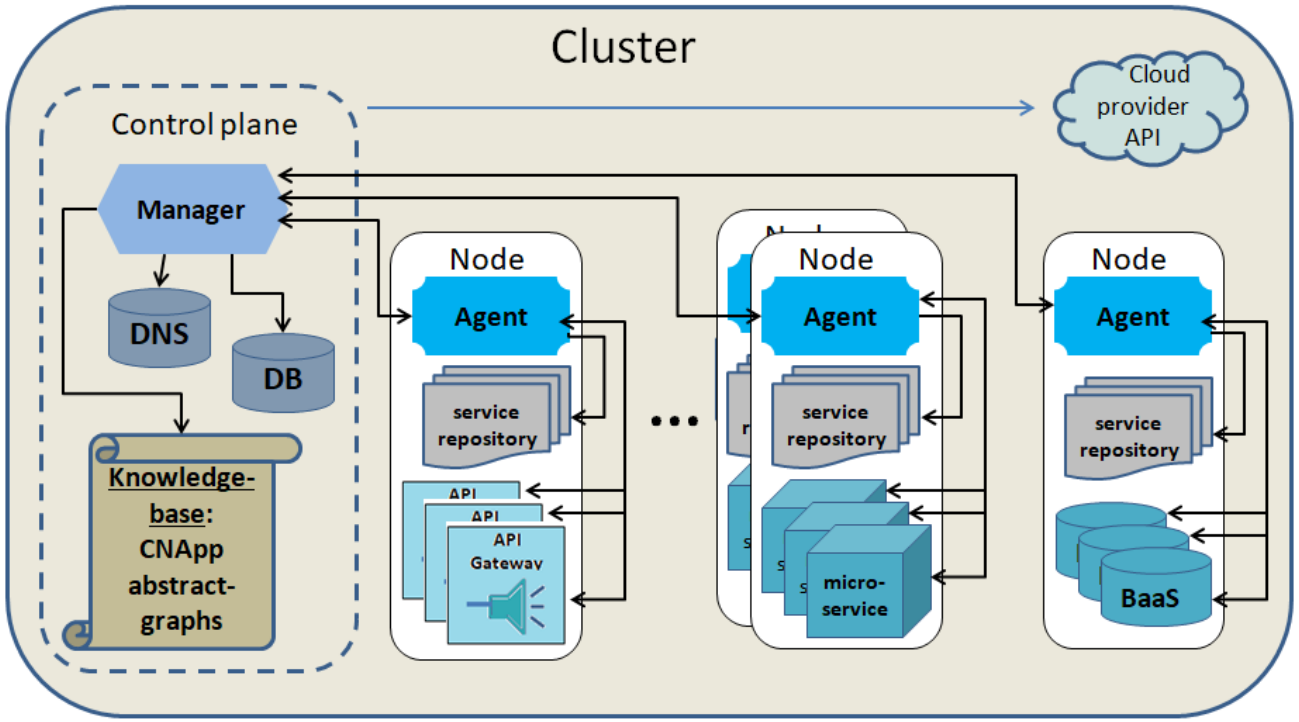
- The main actors of the protocol are: Manager, agents, and running instances of services (API Gateways, regular microservices, and BaaS services)
- There may be two (or more) running instances of the same service. Hence, the term service refers rather to its bytecode.



SSMMP

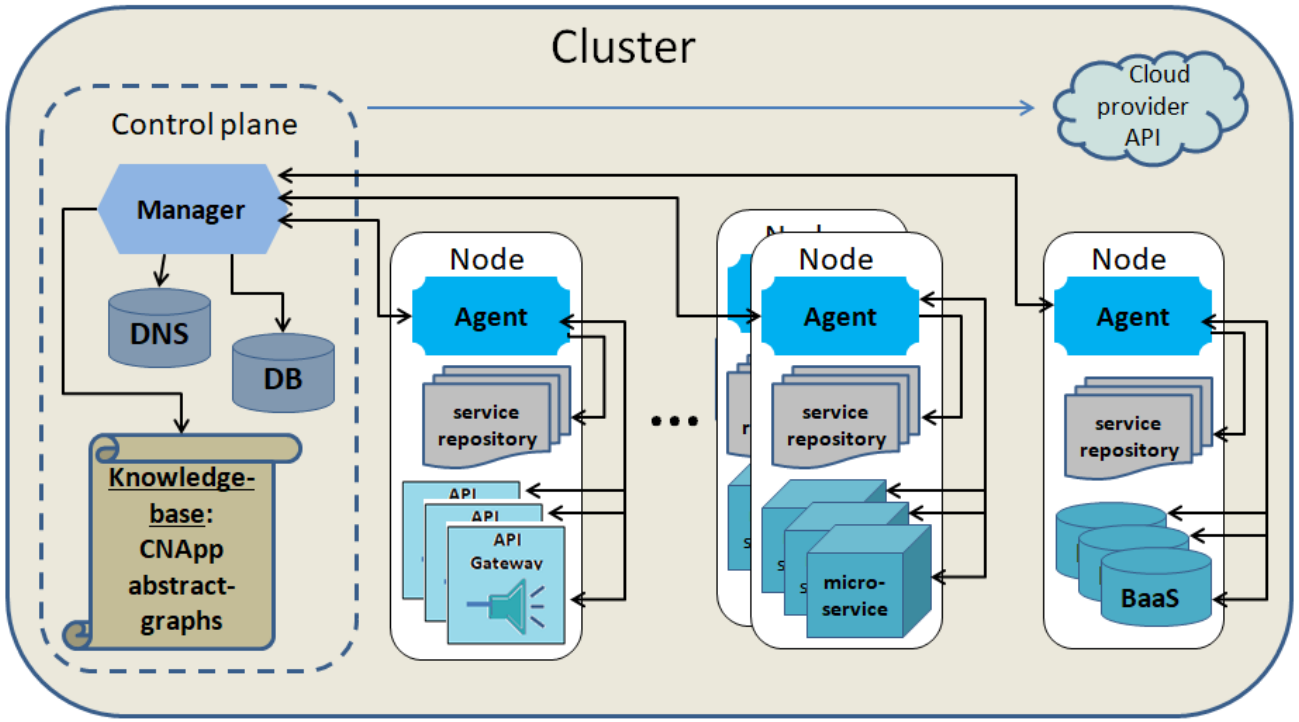
- Manager communicates only with the agents.
- Agent, on a node, communicates with all service instances running on that node.
- Within the framework of SSMMP, any service instance (running on a node) can only communicate with its agent on that node.

- Agent has a service repository at its disposal. It consists of bytecodes of services that can be executed (as service instances) on this node by the agent.



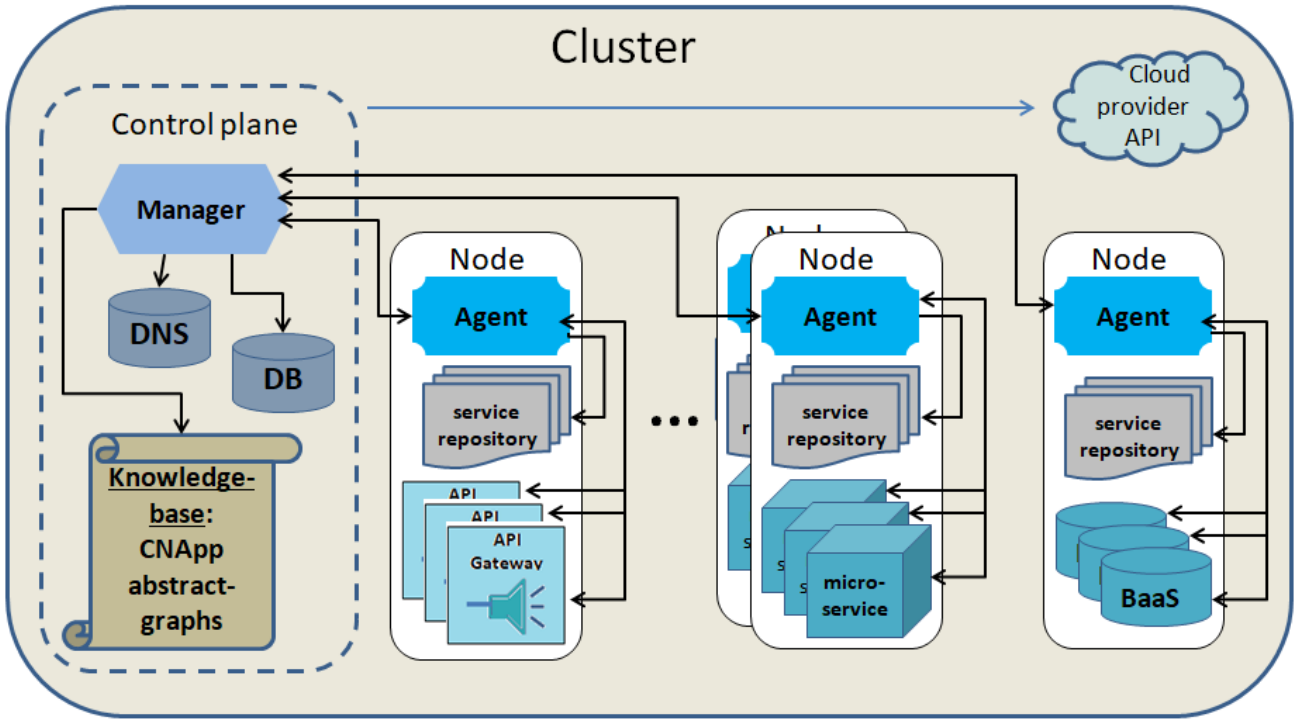
SSMMP

- The agent (as an application) should have operating system privileges to execute applications and to kill application processes.
- Agent acts as an intermediary in performing the tasks assigned by the Manager.
- All service instance executions as well as shutting down running instances are controlled by the Manager through its agents.
- Each agent must register with Manager so that the network address of its node and its service repository are known to Manager.



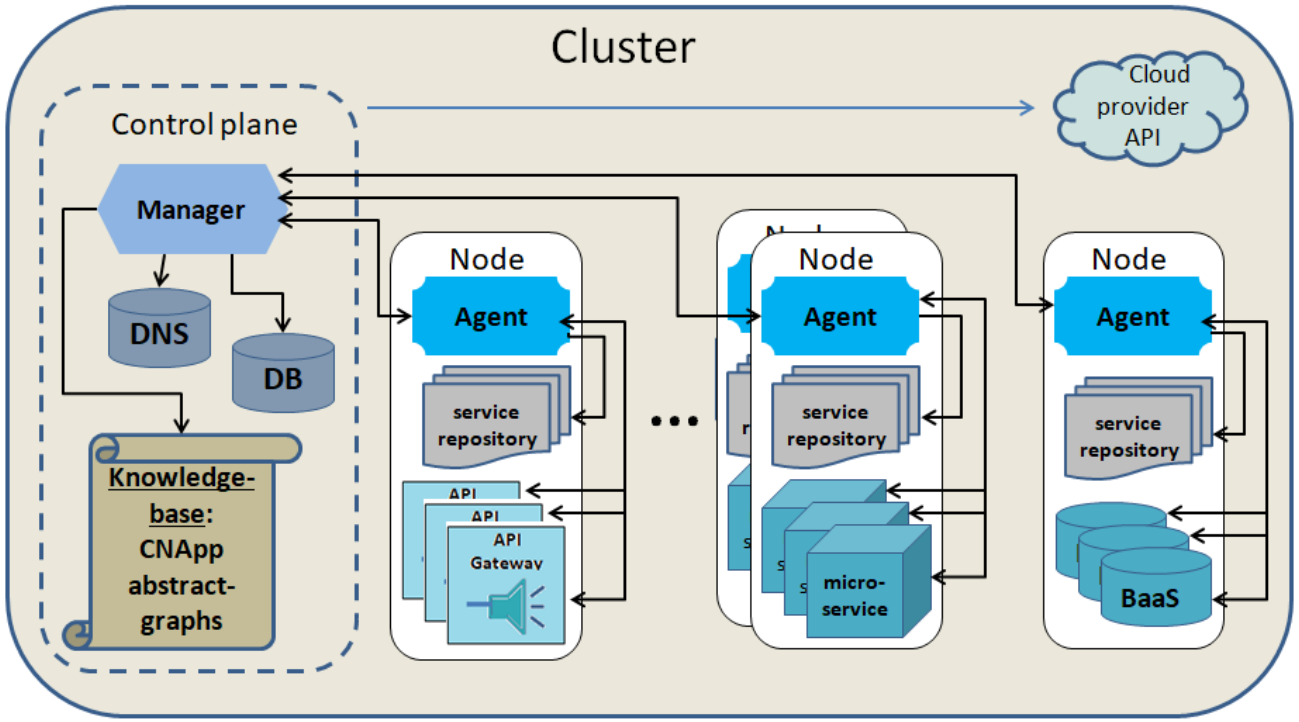
SSMMP

- At a request of Manager, agent can execute instances of services whose bytecodes are available in its repository, or shut down these instances.
- Once a service instance is executed, it initiates the SSMMP communication session with its agent.
- The network address of the agent is, of course, `localhost` for the all service instances running on the same node (host). The port number of the agent (to communicate with its service instances) is fixed for SSMMP, and is the same for all the agents.



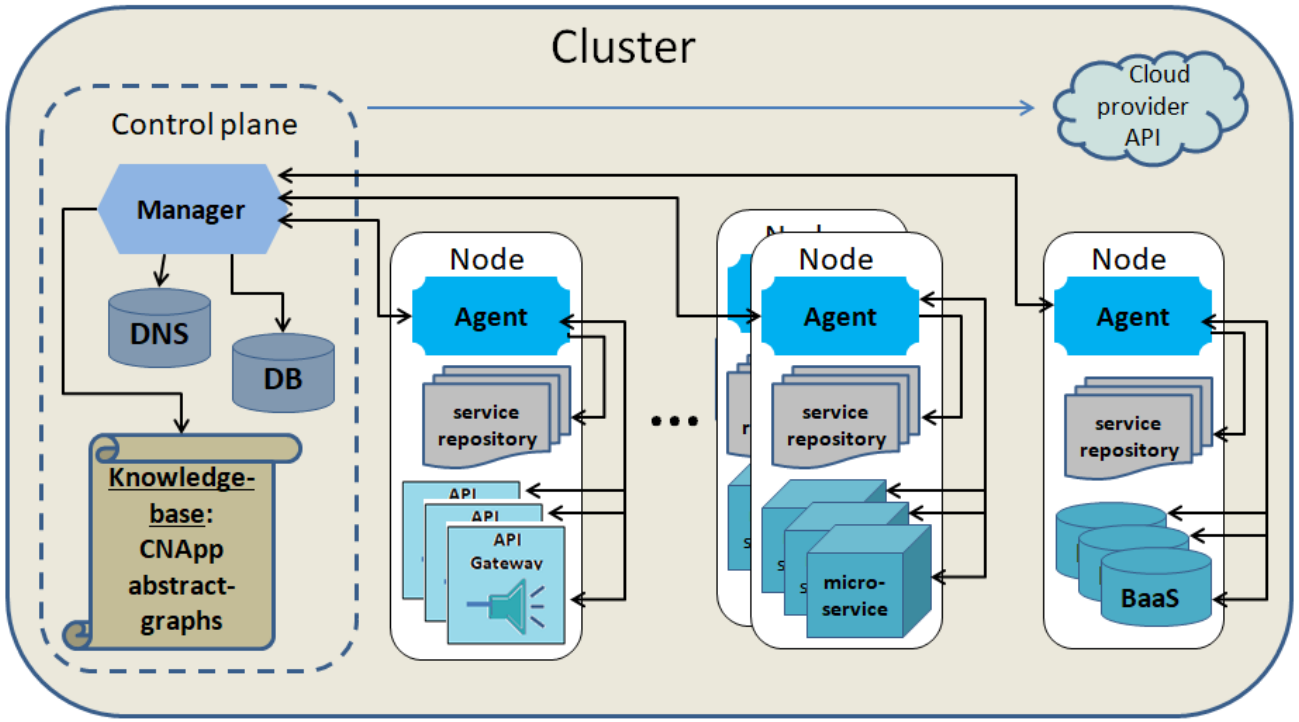
SSMMP

- The agent can monitor the functioning of service instances running on its node (in particular, their communication sessions) and report their status to Manager.
- Manager can also shut down (via its agent) a running instance that is not being used, is malfunctioning, or is being moved to another node.
- Manager controls the execution of CNApps in accordance with the policy defined by the Cloud provider.



SSMMP

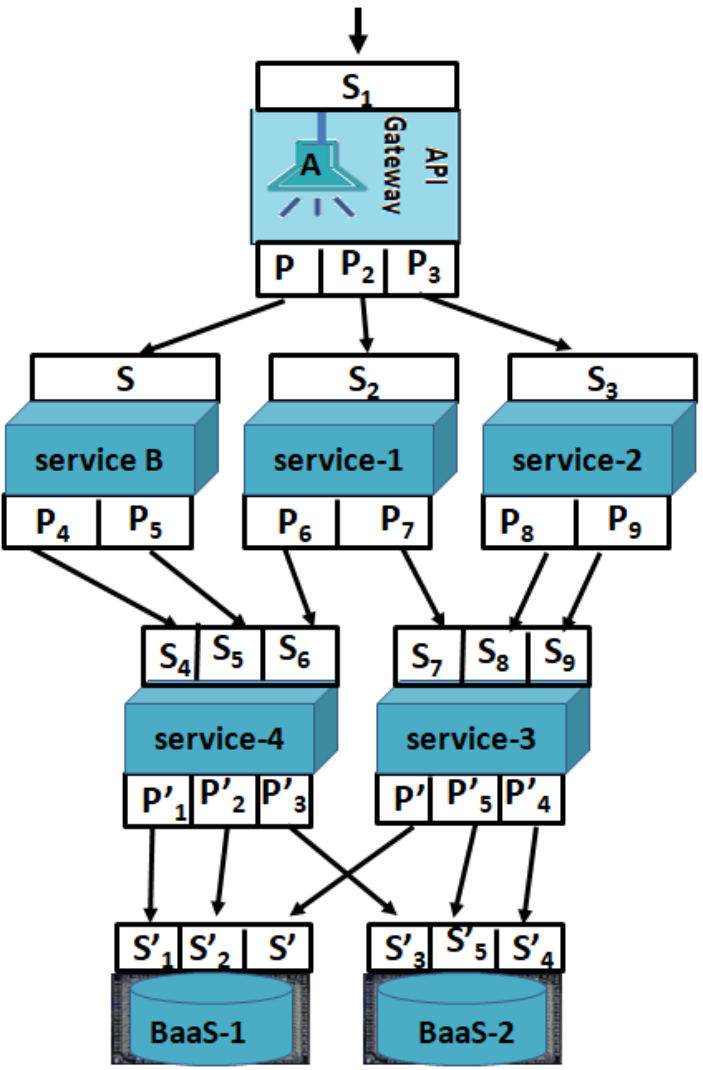
- Current state of the control as well as its history are stored in a dedicated database DB of Manager. Manager knows the service repositories of all its agents.
- The Knowledge-base of Manager consists of abstract graphs of CNApps, i.e. the CNApps that can be deployed on the cluster comprising all the nodes.
- The current state of any running instance of service is stored in Manager's database, and consists of:
  - open communication sessions and their load metrics;
  - observable (healthy, performance and security) metrics, logs and traces.



SSMMP - communication session

•The key element of SSMMP is the concept of *communication session* understood jointly as establishing a connection and then starting a protocol session on this connection.

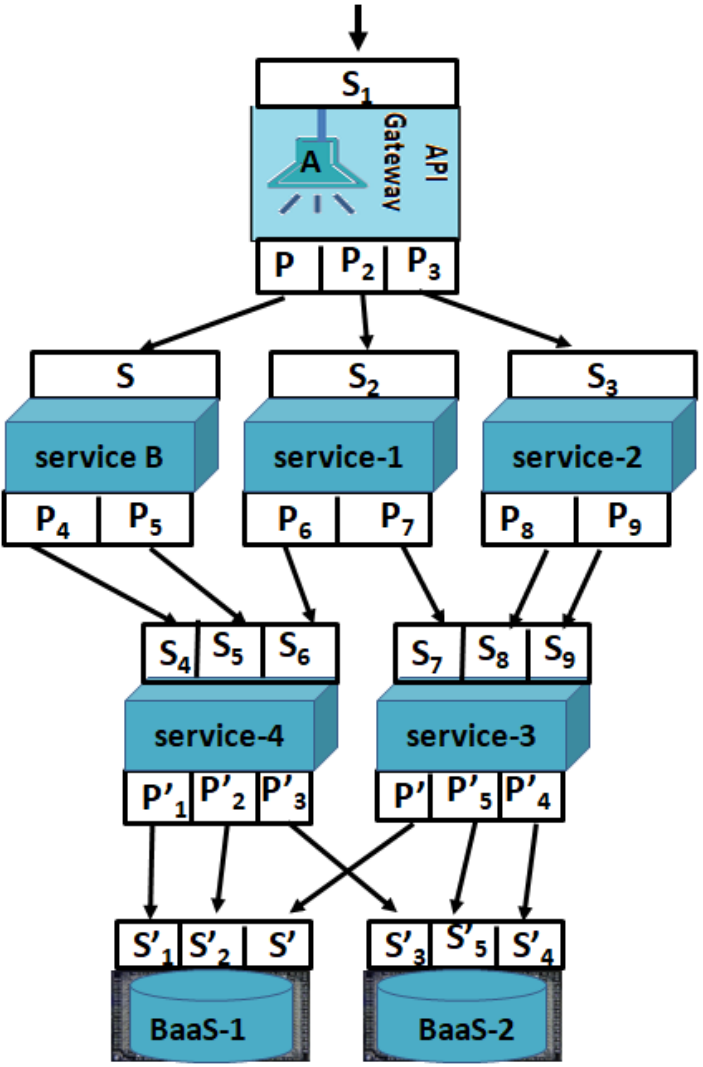
•The process of establishing and closing such sessions is controlled by the Manager through its agents.



- Communication session is an implementation of an abstract connection, say (A, (P,S), B) that is an edge of the abstract graph of CNApp; Figure on the left may serve as an example where A is API Gateway, and B is service B.

The service name B (as a parameter of the abstract connection) is not encoded explicitly in service A. It must be given by Manager as a configuration parameter (according to the abstract graph) for execution of an instance of service A by an agent.

Thus, service A as well as the all services are supposed to be generic, i.e. A may be used as a component of another CNApp for a connection, say (A, (P,S), C), where C is different than B.

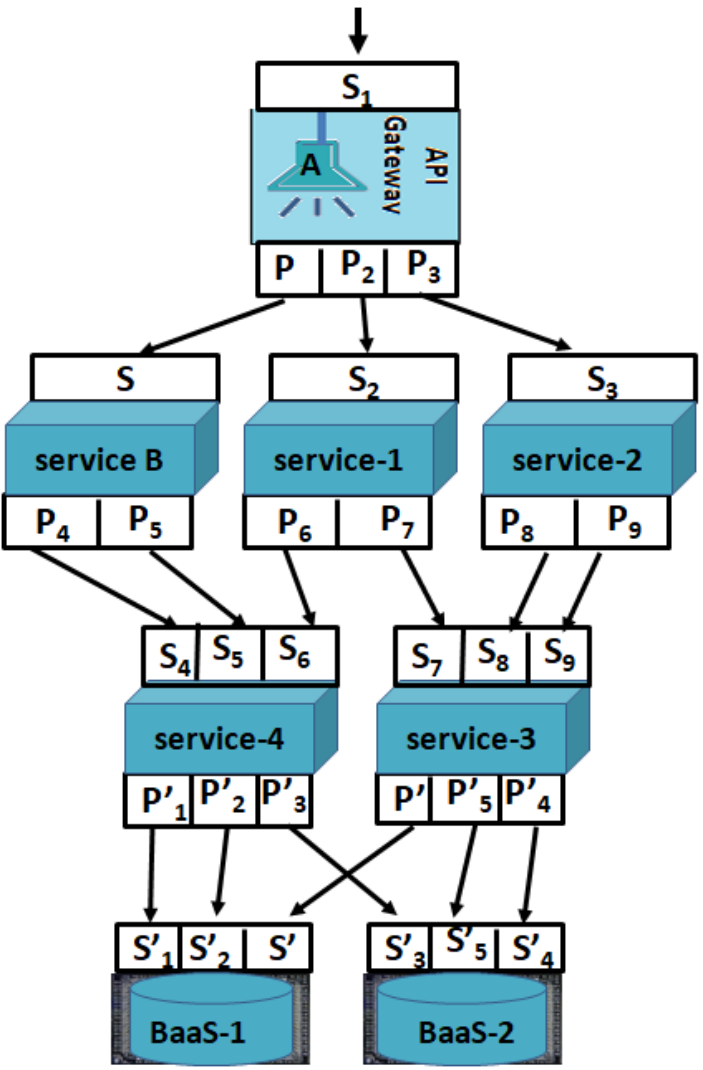


- In order to execute an instance of the service A, the Manager sends a request to an agent (which has the bytecode of A in its repository, and resides on a node with a fixed network address) to execute that bytecode.

The request has parameters for this execution, which include: ports numbers for the socket of A, and service names for its plugs.

For the plug P it is B.

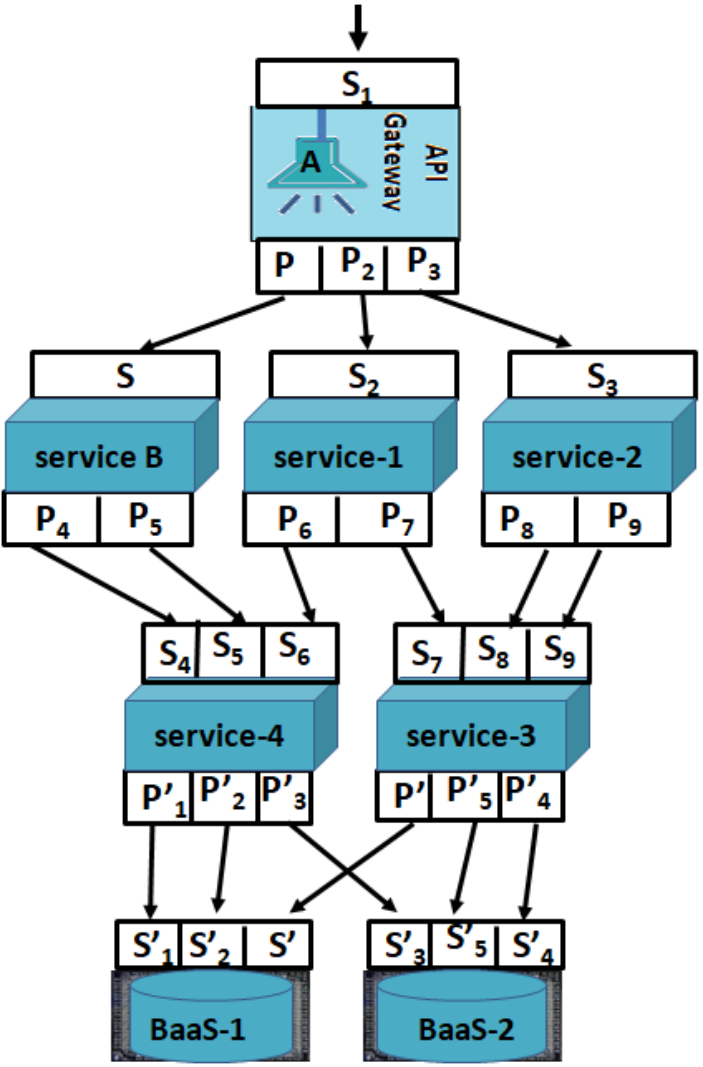
Let an instance of service A (denoted i) be running.



- To implement P (as a TCP client) in the instance i, this instance needs to have a translation of the parameter B to the network address of the host where the instance j of the service B is already running, and the port number on which the socket S (as a TCP server) of the instance j listens to clients.

The port number of this socket, and generally the ports numbers of all sockets of B, are configured by Manager as parameters dedicated to that very instance execution.

This allows multiple instances of the same service to run on the same node (same network address but different port numbers for the sockets).

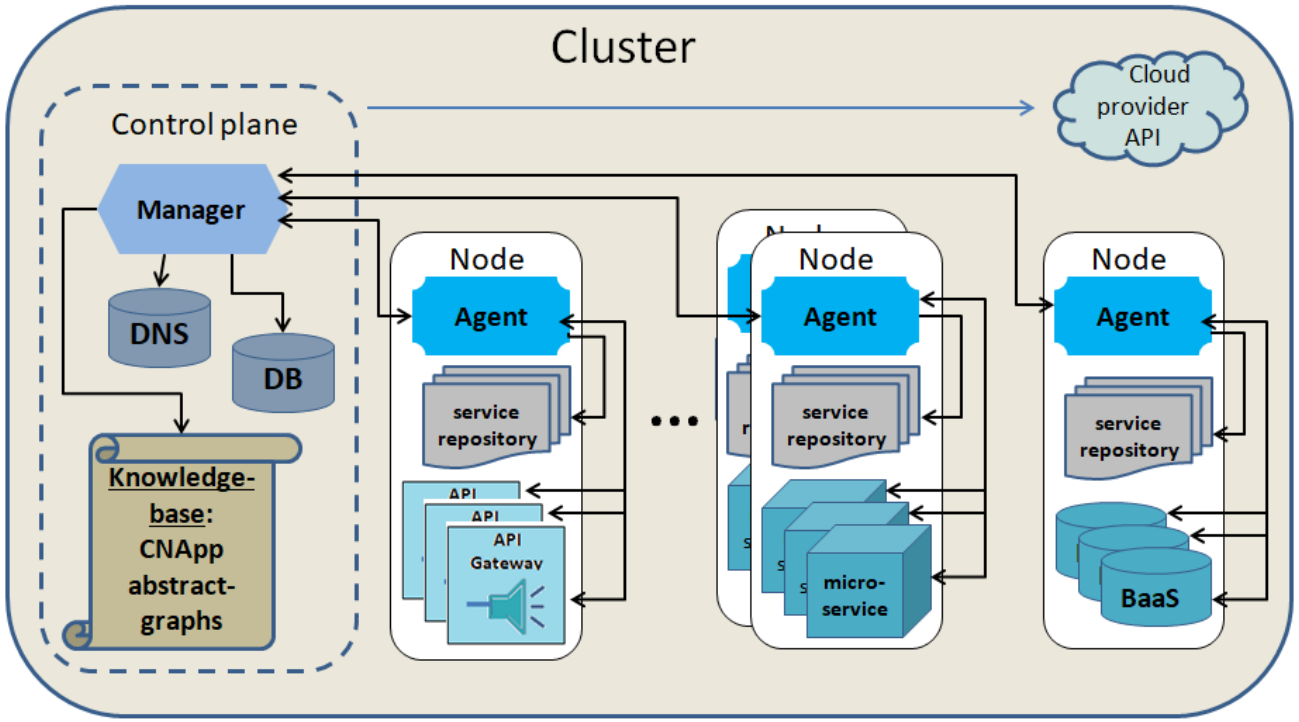


- In order to implement the abstract connection (A, (P,S), B), the instance i sends a request to Manager (via its agent) to translate the parameter B.

  The Manager responses (via the agent) with a translation.

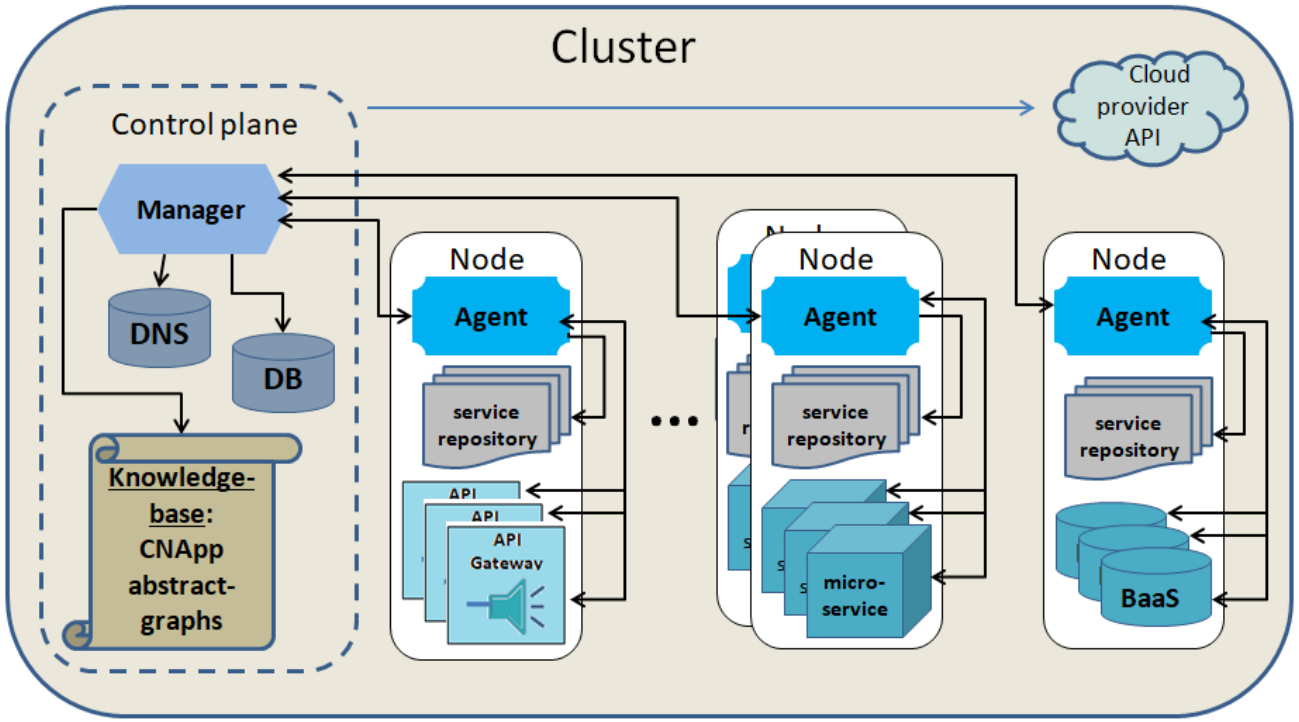
The translation contains the network address of a running instance of B, and the port number of socket S of the instance.

Then a communication session (as an implementation of the abstract connection (A, (P,S), B)) can be established by the instance i.



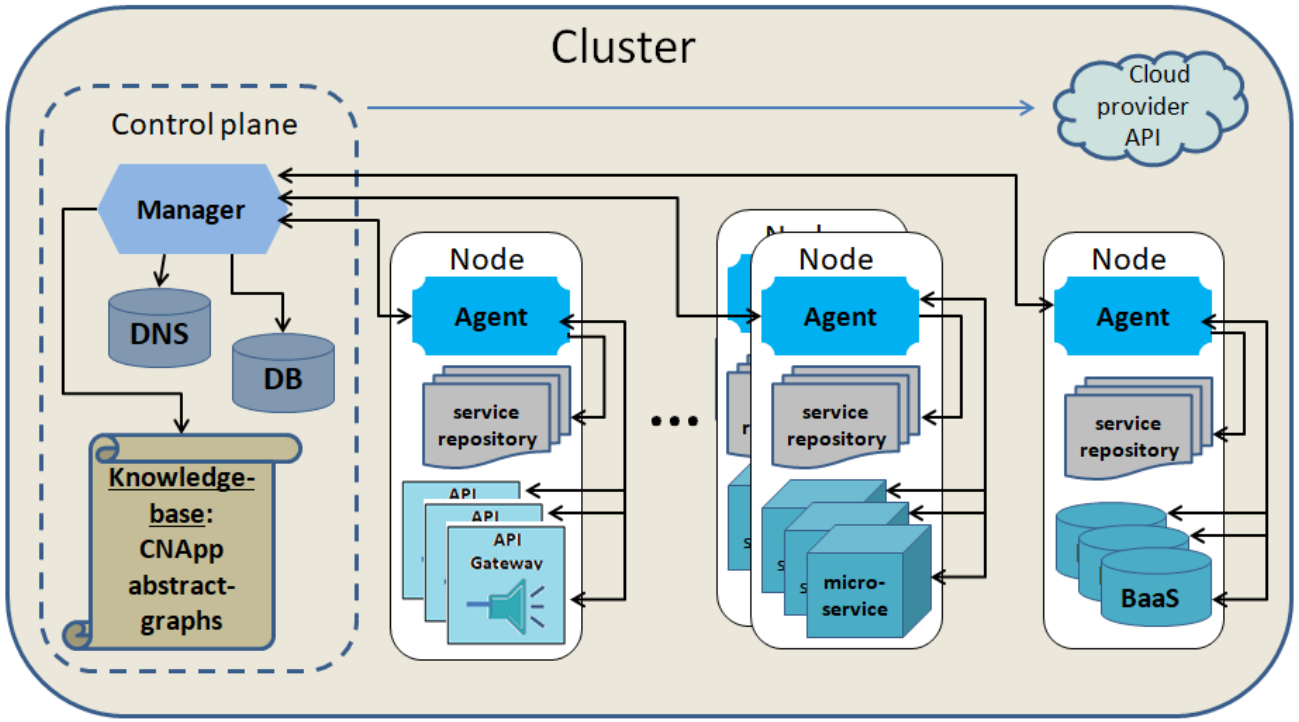
SSMMP - communication session

- If a service instance is running and not in use, there was no reason to start
- it. Therefore, service instances should be started only when they are needed, and shut down when they are no longer needed.
- The same applies to establishing communication sessions. These two aspects are closely related, i.e. if all sessions of a running service instance (which is not an API Gateway) are closed or have not been established for some predetermined period of time, then the instance should be shut down.
- *Current state* of a running CNApp is defined by running service instances, and already established (and not closed) communication sessions between the instances.



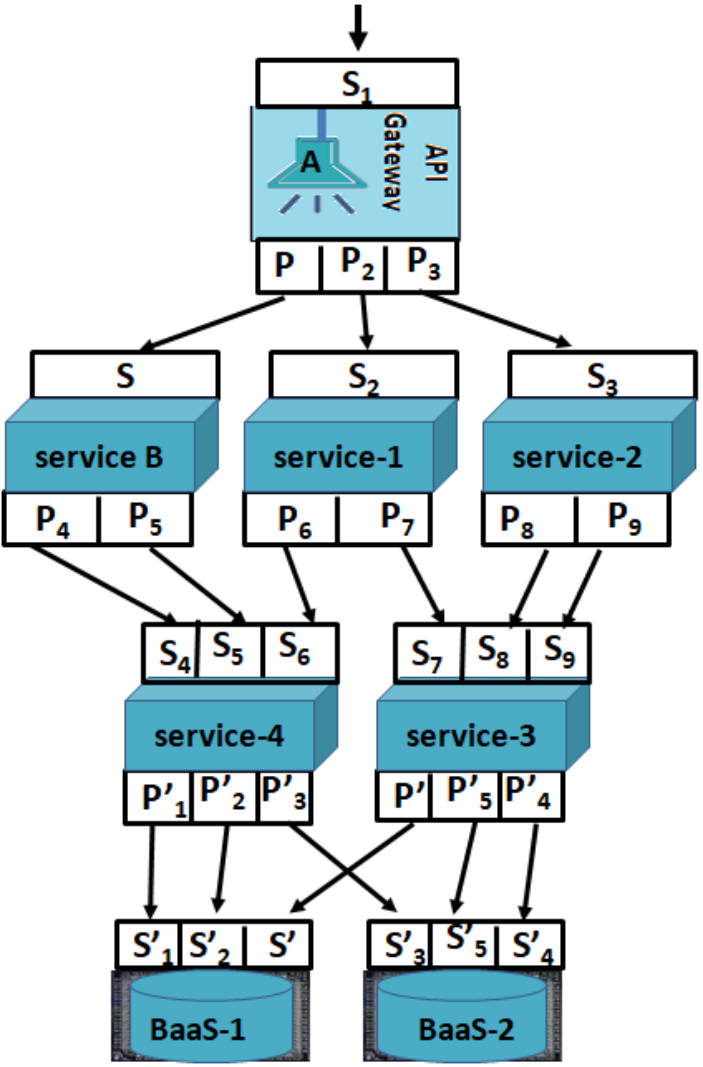
SSMMP - communication session

- *Current state* is the basis for decisions made by Manager:
  - execution and shutdown of service instances,
  - load balancing by multiple instance executions of stateless services, and closing some of them,
  - establishing or closing communication sessions,
  - and reconfiguration of running instances; this includes moving some instances to other nodes.
- These decisions (mutually interrelated) are forwarded to appropriate agents as tasks to be accomplished.



SSMMP - communication session

-



- Usually, API Gateway is the Web interface for users of CNApp.
- In Fig. on the left, socket S_1 implements HTTP server listening at default port number 80.

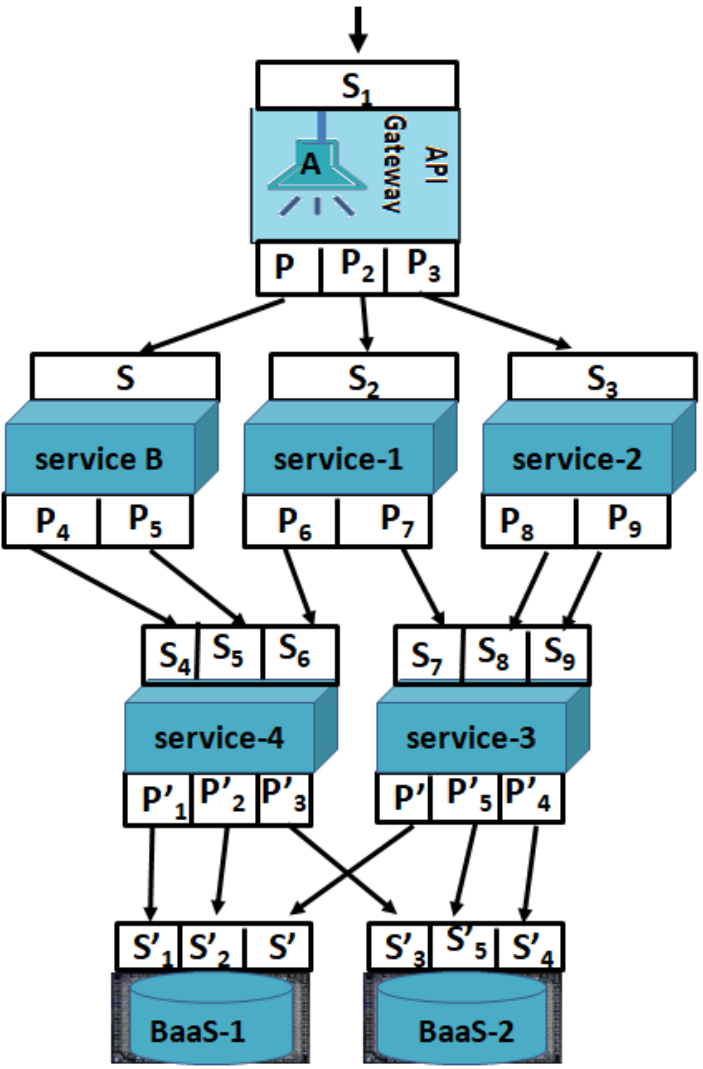
Multiple instances of an API Gateway can be executed (on the basis of DNS load balancing controlled by Manager) so that users requests are distributed across many instances of the API Gateway.

% Manager has a DNS server for load balancing of the services, especially for API Gateways.

This is done in the following way.

The alias (the name) of an API Gateway and the port number (by default it is 80) are supposed to be known to all users of the CNApp.

Execution of an instance of the API Gateway is done in the following way.



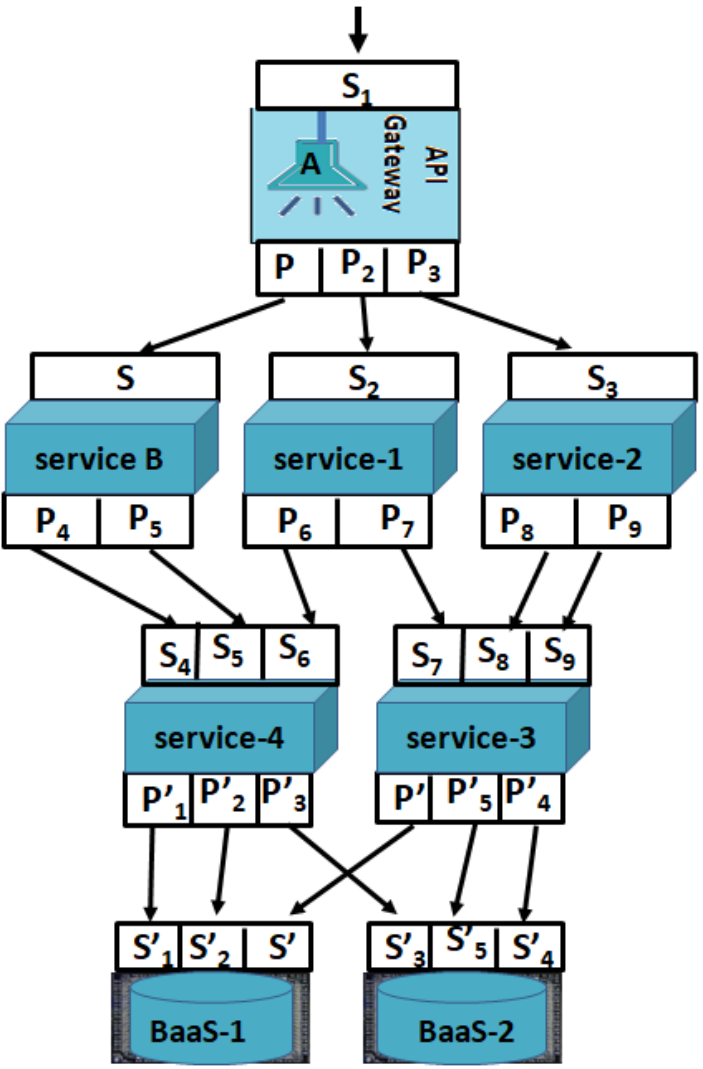
- Manager sends a request to an agent residing on a node to execute an instance of the API Gateway. The request includes a configuration of the plugs of that API Gateway. In Fig. on the left, the plugs are P, P_2, and P_3.

  It is supposed that the bytecode of that API Gateway belongs to the agent's repository.

  The agent executes the instance, and sends the confirmation to Manager.

  Manager stores the instance identifier (as a canonical name) and its network address in its database, and adds two records to its DNS: a record of type A, and a record of type CNAME.

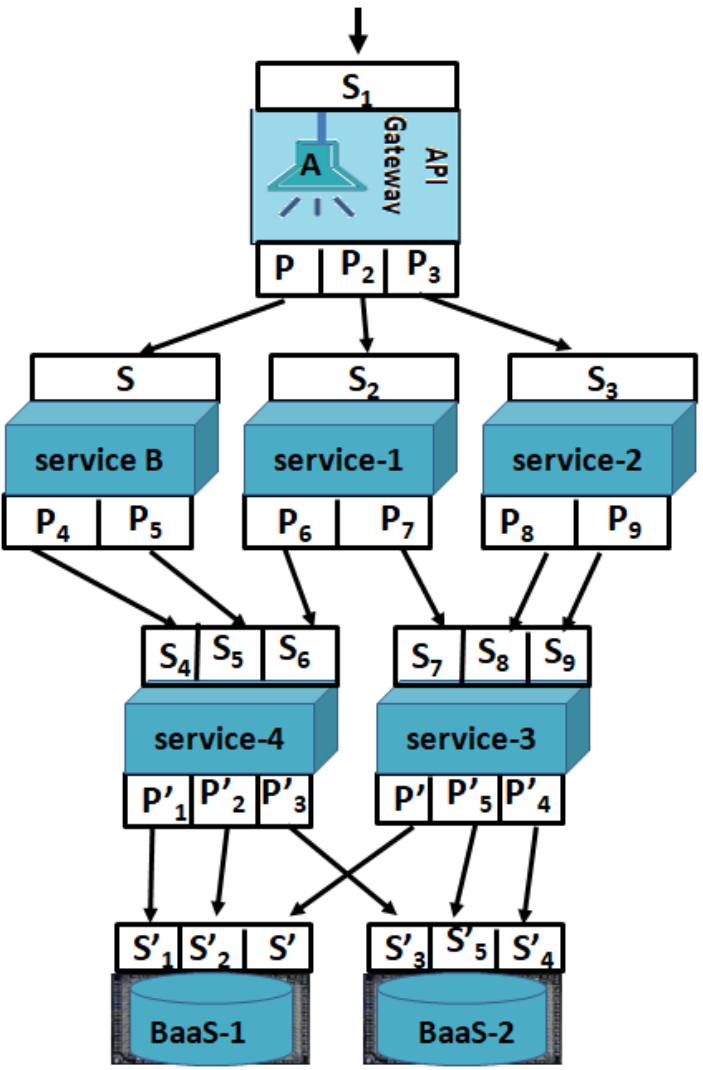
  Thus, a request to resolve the API Gateway name (alias) via DNS is answered by sending the network address of one of the running API Gateway instances.



- Note that no fixed port numbers for sockets are a priori assigned to any regular service or any BaaS service.

The exceptions are API Gateways, where port numbers are fixed and should be well known to users.

Execution of an instance of a regular service or a backend storage service is done as follows. Let us consider service-1 from Fig. on the left, as an example.



- Manager sends a request to an agent residing on a node to execute an instance of service-1.
- It is supposed that the bytecode of service-1 is in the agent's repository.

The request includes the configuration of the port numbers (assigned by Manager) for all sockets of service-1.

In this case, it is one socket S_2.

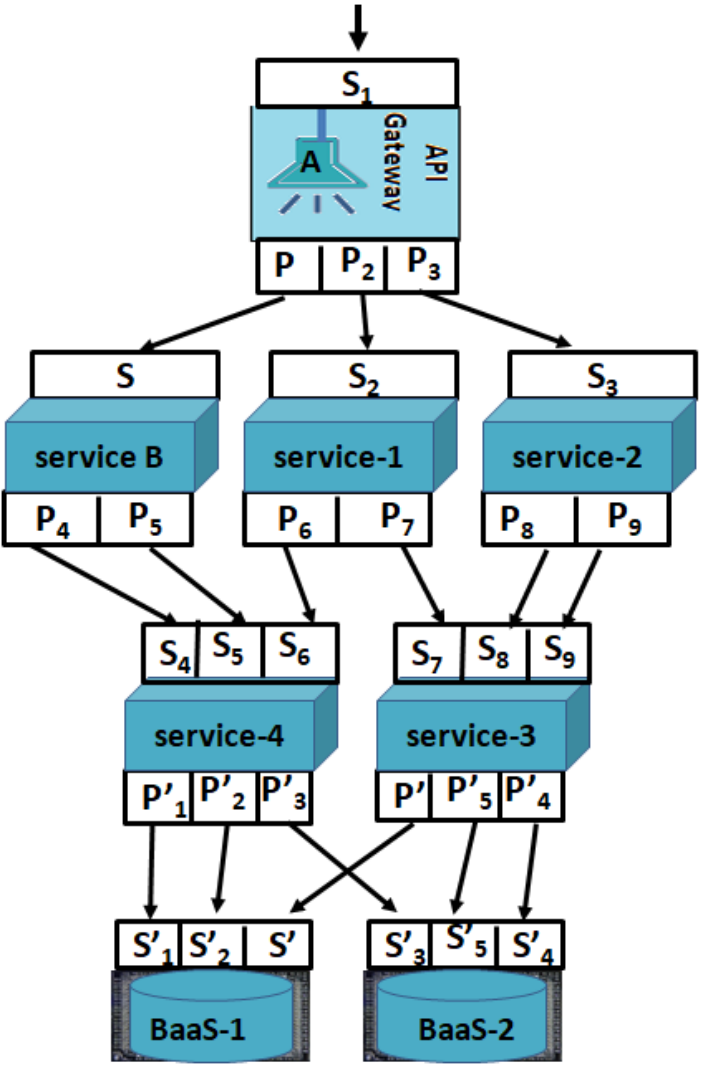
The configuration of the plugs of service-1 (i.e. plugs P_6 and P_7) assigns to each such plug a service name according the abstract graph of the CNApp.

In the Fig., plug P_6 is assigned to service-4 according to the edge

(service-1, (P_6, S_6), service-4).

Similarly, the plug P_7 is assigned to service-3 according to the edge

(service-1, (P_7, S_7), service-3).

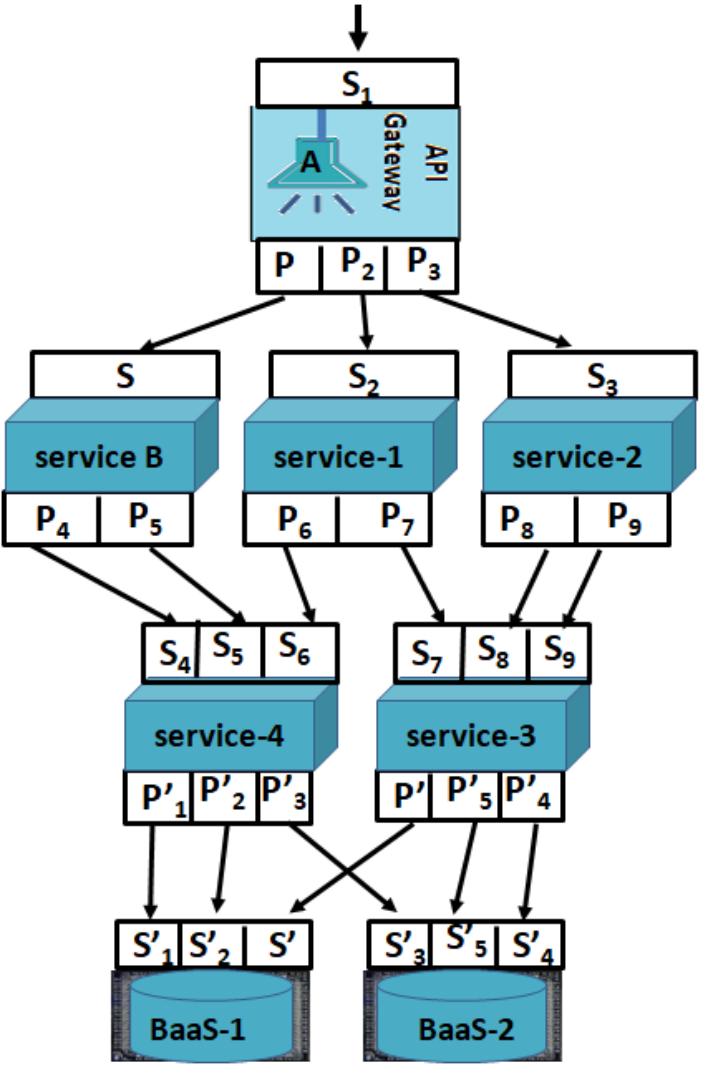


- The agent executes an instance of service-1 for these configurations.

Agent sends to Manager the confirmation of the successful execution.

Manager stores in its database DB the following items: the name of the service, identifier of the instance, the network address of the instance, the configuration of port numbers of the sockets, and the configuration of the plugs.

This is crucial for establishing new communication sessions that this instance will participate in.

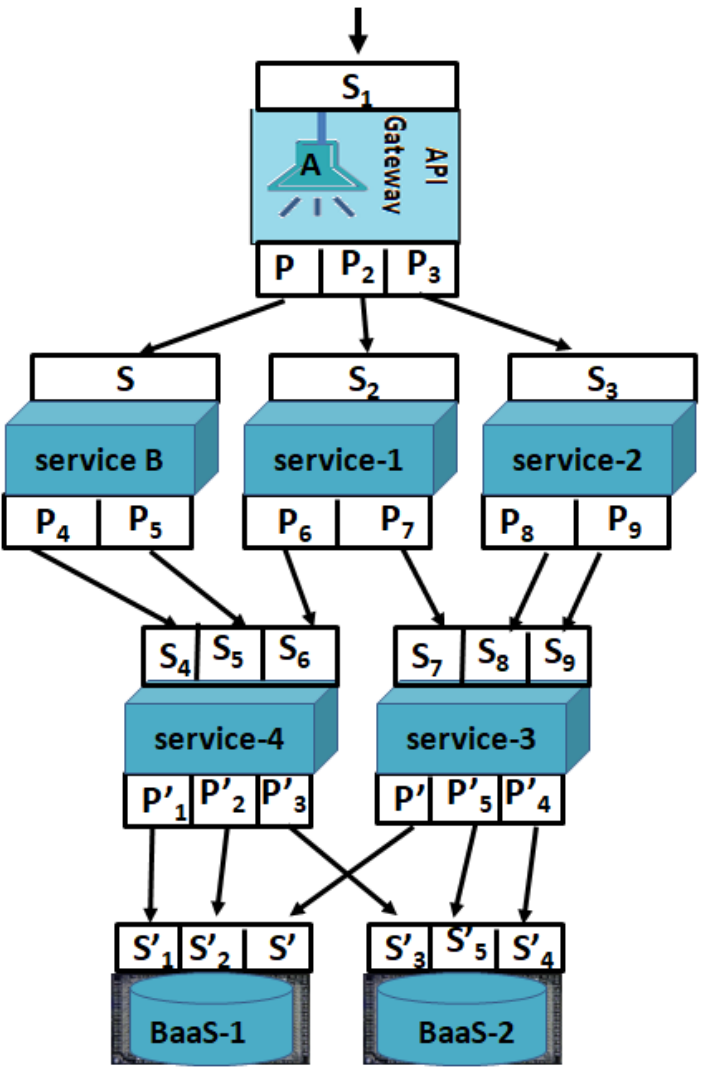


- Establishing a communication session for abstract connection (service-1, (P_6, S_6), service-4) is done in the following way.
- An already running instance i of service-1 sends a request to its agent for: the network address of a node where an instance of service-4 is running, and the port number of the socket S_6 of this instance.

The request is forwarded to Manager that is responsible either to choose (from its DB) a node where an instance of service-4 is already running, or to execute new instance of service service-4 on a node.

Manager sends the port number of S_6 and the network address of a running instance (denoted j) of service-4 to the agent.

Then, the agent forwards it to instance i of service-1 where a concrete plug for the abstract plug P_6 can be construed now, and the communication session can be established to socket S_6 of j. %If TCP/IP is used, this is exactly the establishment of a TCP connection.



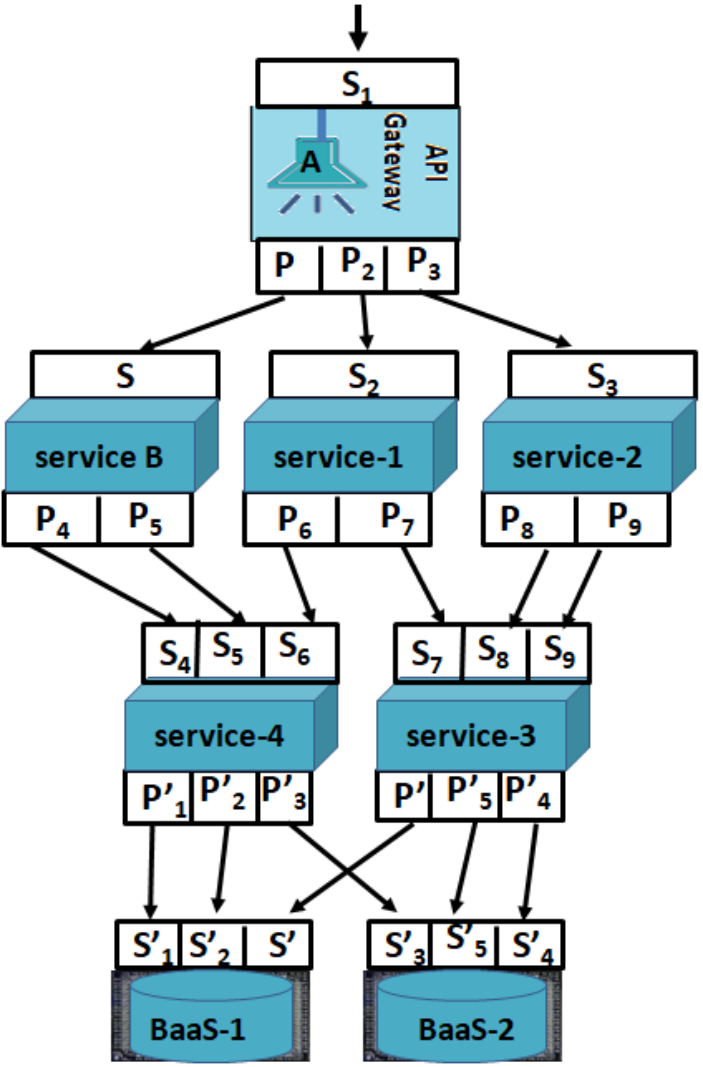
- Closing an existing communication session (between running instance i of C and a running instance j of D) for abstract connection (C, (P,S), D) can be initiated by one of the instances, like for the TCP connection.

Then, the services must report the successful closing to their agents. These reports are forwarded to Manager.

The session closing may be also requested by Manager via the agents.

The agents forward the request to the instances. The instances close the session, and report this to their agents.

The agents forward these reports to Manager.



- The Manager can request the agent to shut down a running service instance.
- For graceful shutdown of a running instance, all its communication sessions should be closed beforehand. Then, an internal method (like `System.exit()` in Java can be evoked to shut down the instance.

Manager can enforce (via its agent) a hard shutdown of a running instance of a service by killing the instance process. This can be done when a running instance fails, e.g. does not respond to its agent or is malfunctioning.

Before this hard shutdown, all communication sessions in which this instance was participated should be closed by participants of opposing sides of the sessions at the request of the Manager through its agents.

If that shutdown instance was for an API Gateway, Manager removes the records (related to that instance) from its DNS.

Summary

- SSMMP is simple if we consider its description presented above, and especially the complete formal specification in the Appendix below.

- The concept of abstract connection between services (in the abstract graph of CNApp) and its implementation as communication sessions is crucial.
- The abstract definition of service of CNApp is also important here.
- Separation of these abstract notions from deployment is important.

- The novelty of SSMMP consists in the dynamic establishment and closing of communication sessions at runtime based on the configurations assigned to sockets and plugs by the Manager.

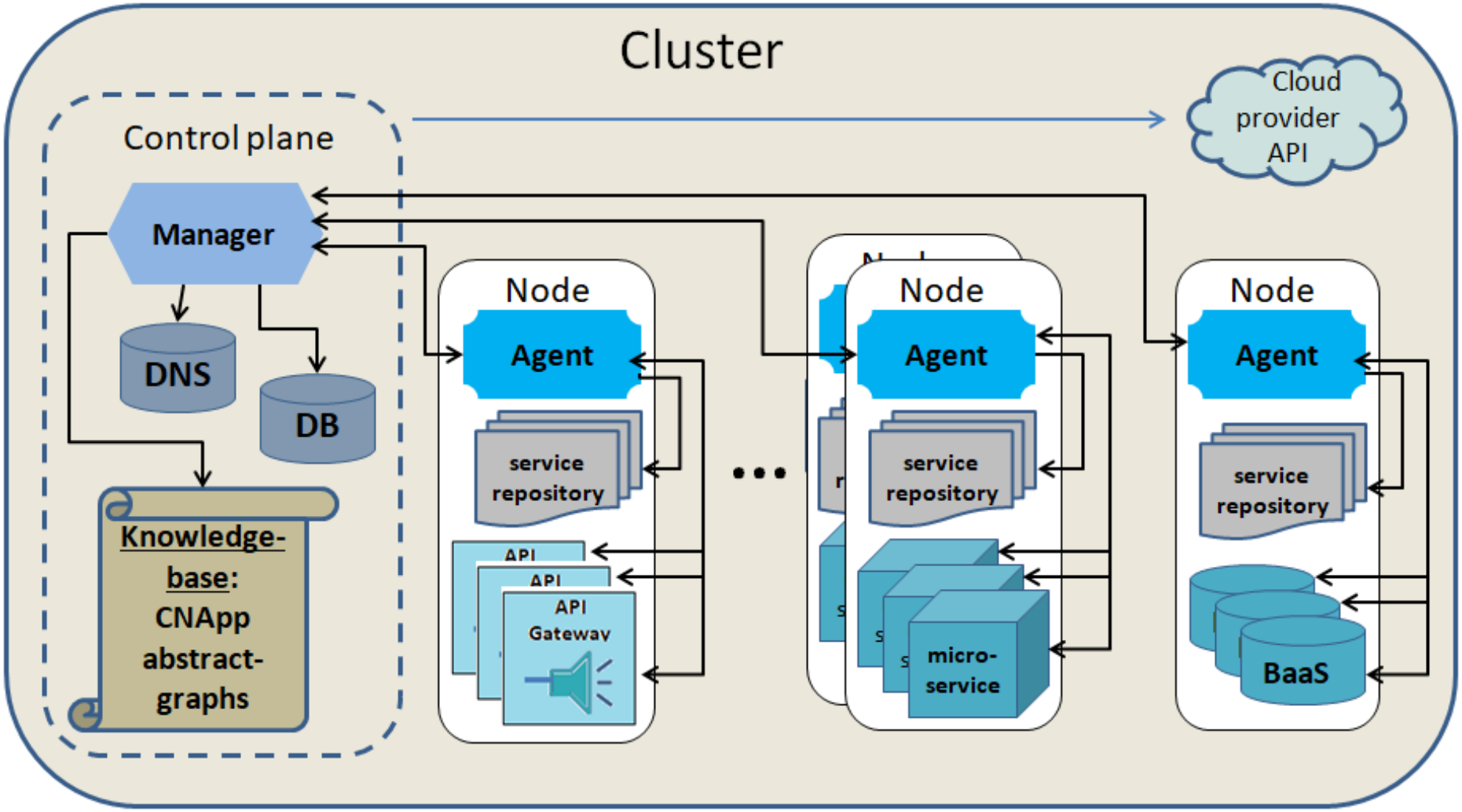
- Although a similar approach has already been used in Netflix (as dedicated software), it can be fully exploited in Netflix by extending the network protocol stack with SSMMP.

Summary

- Since executing, scaling and reconfiguration of CNApp can be done by SSMMP, it seems reasonable to include SSMMP as an integral part of CNApp.
- Then, also CNApp crash recovery can be performed via SSMMP.
- Graph of CNApp and the states of its running instances are stored by Manager in its KB and DB.
- Failures of agents and service instances can be handled if Manager is running properly.
- Replications of cluster nodes, agents and their service repositories are sufficient means for recoveries from such failures.
  The central Manager is the weakest point here; its failure results in an irreversible failure of the running CNApp.

Summary

- However, if the Manager's current state is kept securely by a supervising manager, the Manager process can be recovered from that state.
- The supervising manager can also serve as a distributed control plane where there are several Managers, each controlling a portion of the CNAApp abstract graph.
- Netflix uses over 1000 microservices now. Uber now has 4000 or more independent microservices. A distributed control plane is necessary for such huge CNApps.
- SSMMP was designed to be independent of transport and network protocol stack. TCP/IP is the default stack for communication sessions. Named Data Networking may be seen as an alternative.


Cluster
Control plane
Manager
DNS
DB
Knowledge-base: CNApp abstract-graphs
Cloud provider API
Node
Agent
service repository
API
API
API Gateway
Node
Agent
service repository
micro-service
Node
Agent
service repository
BaaS

APPENDIX: SSMMP - specification of the protocol messages and actions

- normal (not italic) letters to denote services, their instances, plugs, sockets and connections, like A, P, S, B, i, and j.
- There are two general kids of messages: request, and response to this request.
- All messages are strings.
- Message consists of a sequence of lines.
- Line is of the form:

```
line_name: contents
```

- The first line is for message type.
- The second line is for message identifier (an integer). The identifier is unique, and is the same for request and its response.

Initialization of the protocol

- Let a CNApp be fixed. The abstract graph of CNApp is known to Manager. For each service of CNApp, there are agents that can execute instances of this service, i.e. the bytecode of this service belongs to the agents repositories.

- Each entry of the repository is of the form (service-name, list of socket names, list of plug names, bytecode of service).

- An agent registers with Manager and sends the list of service names of its repository.
- Request for the registration from agent to Manager is of the following form.

Request for the registration from agent to Manager

```
type: initiation_request
message_id: [integer]
agent_network_address: [IPv6]
service_repository: [service name list]
```

In square brackets of

```
message_id: [integer]
```

there is an element of a datatype, in this case it is a positive integer determined by the agent.

For `agent_network_address: [IPv6]`

it is a concrete IPv6 network address.

For the line

```
service_repository: [service name list]
```

the contents denotes a sequence of the form

```
(name1; name2; … name_k)
```

Registration response form Manager to agent

```
type: initiation_response
message_id: [integer]
status: [status code]
```

- Universal HTTP response status codes are proposed to be adapted to SSMMP. Their meaning depends on response types.
- Each code consists of three digits, and is of the form:

- `1xx informational response`
- `2xx successful` – the request was successfully received, understood, and accepted
- `3xx redirection` – further action needs to be taken in order to complete the request
- `4xx requester error` – the request contains bad syntax or cannot be fulfilled
- `5xx respondent error`

Execution of service A

- Manager assigns a unique identifier, say i, (a positive integer) to a new instance of A to be executed. Manager also determines port numbers to all sockets of the instance. It is called socket configuration, and is a sequence of pairs:

  `(socket_name, port_number)`

- Configuration of plugs is a sequence of pairs:

  `(plug_name, service_name)`

- assigned to instance i by Manager according to the CNApp abstract graph.
- This means that each abstract plug is assigned a service name, where is the corresponding abstract socket.
- Manager also determines a network address (denoted `NA_i` of the node where the instance i of A is to be executed by the agent residing on that node. % It is also the network address of the agent.

Request from Manager to the agent to execute the instance i of service A

```
type: execution_request
message_id: n
agent_network_address: NA_i
service_name: A
service_instance_id: i
socket_configuration: [configuration of sockets]
plug_configuration: [configuration of plugs]
```

- Action of the agent: execution of instance i of the service A for these configurations of sockets and plugs.
- Response from agent to Manager

```
type: execution_response
message_id: n
status: [status code]
```

Communication session establishment

- Establishing a communication session for abstract connection (A, (P,S), B) between instance i of A, and instance j of B.
- Let us suppose that instance i of service A is already running on the node that has network address `NA_i`
- Request from the instance i of service A to its agent:

```
type: session_request
message_id: n
sub_type: service_to_agent
source_service_name: A
source_service_instance_id: i
source_plug_name: P
dest_service_name: B
dest_socket_name: S
```

Communication session establishment

Request is forwarded to Manager by the agent:

```
type: session_request
message_id: n
sub_type: agent_to_Manager
agent_network_address: NA_i
source_service_name: A
source_service_instance_id: i
source_plug_name: P
dest_service_name: B
dest_socket_name: S
```

Communication session establishment

- If there is no instance of service B already running, then
- Manager sends a request to an agent to execute an instance j of service B.

- Otherwise, i.e. if instance j of service B (on the node with network address `NA_j` and the port k for S) is already running, then Manager sends the following response to the agent:

```
type: session_response
message_id: n
sub_type: Manager_to_agent
status: [status code]
dest_service_instance_network_address: NA_j
dest_socket_port: k
```

Communication session establishment

- Response from agent to instance of A:

```
type: session_response
message_id: n
sub_type: agent_to_service
status: [status code]
dest_service_instance_network_address: NA_j
dest_socket_port: k
```

Communication session establishment

- Action of instance i of A: initialize (P,S) session to instance j of B. The port number of plug P is determined; let it be denoted by m.
- By default, the socket S of instance j of B accepts the session establishment. This acceptance will be known to Manager, if the session acknowledgment is send by instance i to Manager via the agent.
- Action of instance j of B: accept the establishing (P,S) session to instance i of A.
- New socket port number (say l) is assigned to this session; this is exactly the same as for TCP connection.
- Instance j of B gets to know the values of the parameters:

```
source_service_instance_network_address: NA_i
source_plug_port: m
```

- Instance i of A gets to know the value of the parameter

```
dest_socket_new_port: l
```

Communication session establishment

- Acknowledgment of the established session is sent by the instance i of A to its agent.

```
type: session_ack
message_id: n
sub_type: service_to_agent
status: [status code]
source_plug_port: m
dest_socket_new_port: l
```

Communication session establishment

and forwarded to Manager by the agent:

```
type: session_ack
message_id: n
sub_type: agent_to_Manager
status: [status code]
source_plug_port: m
dest_socket_new_port: l
```

- Note that `message_id` is n (determined by instance i of A), and is the same for all the above request, response and acknowledgment messages.

Communication session establishment

- The complete list of the parameters of the session is as follows.

```
source_service_name: A
source_service_instance_network_address: NA_i
source_service_instance_id: i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_service_instance_id: j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

Communication session establishment

- The number k is the port number (assigned by Manager) to the socket S (of instance j of B) for listening to clients.
- New port l of the socket S is dynamically assigned by instance j of B solely for the communication session with P of instance i of A.

- The instance i of A knows the above parameters except

```
dest_service_instance_id: j
```

- The instance j of B knows the above parameters except

```
source_service_instance_id: i
source_service_name: A
```

- Manager knows all parameters of the session.

Closing a communication session

(P,S) between running instance i of A and instance j of B

- Each of the instances can initialize session closing, like in TCP connection, according to its own business logic.
- If instance i does so, it informs instance j of B that does the same; and vice versa.
- It is a regular closing of the session.

- A session may be closed only by one part of the communication due to failure of the other part or a broken link making the communication between these two parts impossible.
- In any of these cases above a running instance sends a message to its agent informing that the session was closed.
- Then, the agent forwards it to Manager.

Session closing by instance i of A, or by instance j of B

- The message from instance i of A to its agent is as follows.

```
type: source_service_session_close_info
message_id: n
sub_type: source_service_to_agent
source_service_name: A
source_service_instance_network_address: NA_i
source_service_instance_id: i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

- The instance i of A sends all (known to it) parameters of the session.

Session closing by instance i of A, or by instance j of B

- The agent forwards the info to Manager:

```
type: source_service_session_close_info
message_id: n
sub_type: agent_to_Manager
source_service_name: A
source_service_instance_network_address: NA_i
source_service_instance_id: i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

- The value of the parameter `message_id: n` is the same for the both messages above, and is determined by instance i of A.
- Manager can determine the identifier j of the instance of B on the basis of the port numbers: m, k and l.

Session closing by instance i of A, or by instance j of B

- Session closing by instance j of B is similar.
- The message from instance j of B to its agent is as follows.

```
type: dest_service_session_close_info
message_id: o
sub_type: dest_service_to_agent
source_service_instance_network_address: NA_i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_service_instance_id: j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

- The instance j of B sends all (known to it) parameters of the session.

Session closing by instance i of A, or by instance j of B

- The agent forwards the info to Manager.

```
type: dest_service_session_close_info
message_id: o
sub_type: agent_to_Manager
source_service_instance_network_address: NA_i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_service_instance_id: j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

- The value of the parameter `message_id: o` is the same for both messages above and is determined by instance j of B.
- Manager can determine the identifier i of the instance of A on the basis of the port numbers: m, k and l.

Session closing on the request of Manager

- An initiation of a communication session for abstract connection (A, (P,S), B) between instance i of A, and instance j of B is done by the instance i according to its business logic. Upon the request of instance i, configuration for such session is sent to instance i by Manager via the agent of instance i.

- By default, the socket S of instance j of B accepts the session establishment. This acceptance is known to Manager.

- Manager's request to close this session is only sent to the instance i of service A via the agent of instance i.
- The request contains all parameters of the session known to the instance i of A, i.e. except

```
dest_service_instance_id: j
```

Session closing on the request of Manager

- Request, from Manager to instance i of A via its agent to close a session, is as follows. The value o of the parameter `message_id:` is determined by Manager.

```
type: source_service_session_close_request
message_id: o
sub_type: Manager_to_agent
source_service_name: A
source_service_instance_network_address: NA_i
source_service_instance_id: i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

Session closing on the request of Manager

- The agent forwards the request to instance i of A

```
type: source_service_session_close_request
message_id: o
sub_type: agent_to_source_service
source_service_name: A
source_service_instance_network_address: NA_i
source_service_instance_id: i
source_plug_name: P
source_plug_port: m
dest_service_name: B
dest_service_instance_network_address: NA_j
dest_socket_name: S
dest_socket_port: k
dest_socket_new_port: l
```

- Action of instance i of A: closing P.

Session closing on the request of Manager

- Response of instance i of A to its agent:

```
type: source_service_session_close_response
message_id: o
sub_type: source_service_to_agent
status: [status code]
```

- Forwarding the response to Manager.

```
type: source_service_session_close_response
message_id: o
sub_type: agent_to_Manager
status: [status code]
```

- After the successful session closing, the instance i may need a new communication session (for the same connection) to complete the task interrupted by the enforced closing. In order to do so the instance i can send a request to Manager for a configuration needed to establish such session. The message format of this request is given in a previous slide

- In the case of failure of instance i of service A, or its agent or node, a similar request must be sent to instance j of service B to close the session. This requires only minor modifications to the message sequence above.

Shutdown of service instances

- Graceful shutdown of a running instance of service by itself (on a request of Manager forwarded by agent) can be done after closing of all its communication sessions on the request of Manager via agent. The appropriate requests and responses are as follows.
- From Manager to agent:

```
type: graceful_shutdown_request
message_id: o
sub_type: Manager_to_agent
service_name: A
service_instance_id: i
```

- From agent to service instance:

```
type: graceful_shutdown_request
message_id: o
sub_type: agent_to_service_instance
service_name: A
service_instance_id: i
```

Shutdown of service instances

- Instance i of service A invokes internal method to shut down itself. Just before completing it, the instance sends the following response to the agent.

```
type: graceful_shutdown_response
message_id: o
sub_type: service_instance_to_agent
status: [status code]
```

- The agent forwards the response to Manager:

```
type: graceful_shutdown_response
message_id: o
sub_type: agent_to_Manager
status: [status code]
```

Shutdown of service instances

- Hard shutdown of instance i of service A is done by agent on the request of Manager.

```
type: hard_shutdown_request
message_id: n
sub_type: Manager_to_agent
service_name: A
service_instance_id: i
```

- Action of the agent: kill the process of service instance i.
- The response is as follows.

```
type: hard_shutdown_response
message_id: n
sub_type: agent_to_Manager
status: [status code]
```

Simple monitoring of service instances by agent

- Agent's request for observable metrics from a service instance is as follows.

```
type: health_control_request
message_id: o
sub_type: agent_to_service_instance
service_name: A
service_instance_id: i
```

Simple monitoring of service instances by agent

Service instance response:

```
type: health_control_response
message_id: o
sub_type: service_instance_to_agent
service_name: A
service_instance_id: i
status: [status code]
```

- The status codes are to express the metrics.
- Agent forwards the response to Manager only in the case of abnormal behavior (marked with a status code) of instance i.

```
type: health_control_response
message_id: o
sub_type: agent_to_Manager
service_name: A
service_instance_id: i
status: [status code]
```

Final remarks

- Status codes can be used to handle failures. This is left to SSMMP implementations.
- Requirements for developing services of CNApp participating in SSMMP are as follows.
- Each instance of a service of CNApp is obliged to close its communication session at the request of the Manager. This can interfere with the business logic of the instance.
- For this reason, the current state of the communication session (until closed) must be stored in a BaaS service. To continue a task interrupted by the closing, the instance (at the client side of the connection) can establish a new session for the same abstract connection to continue and possibly to complete the task. Retrieval of the current state from the BaaS service may also be needed.

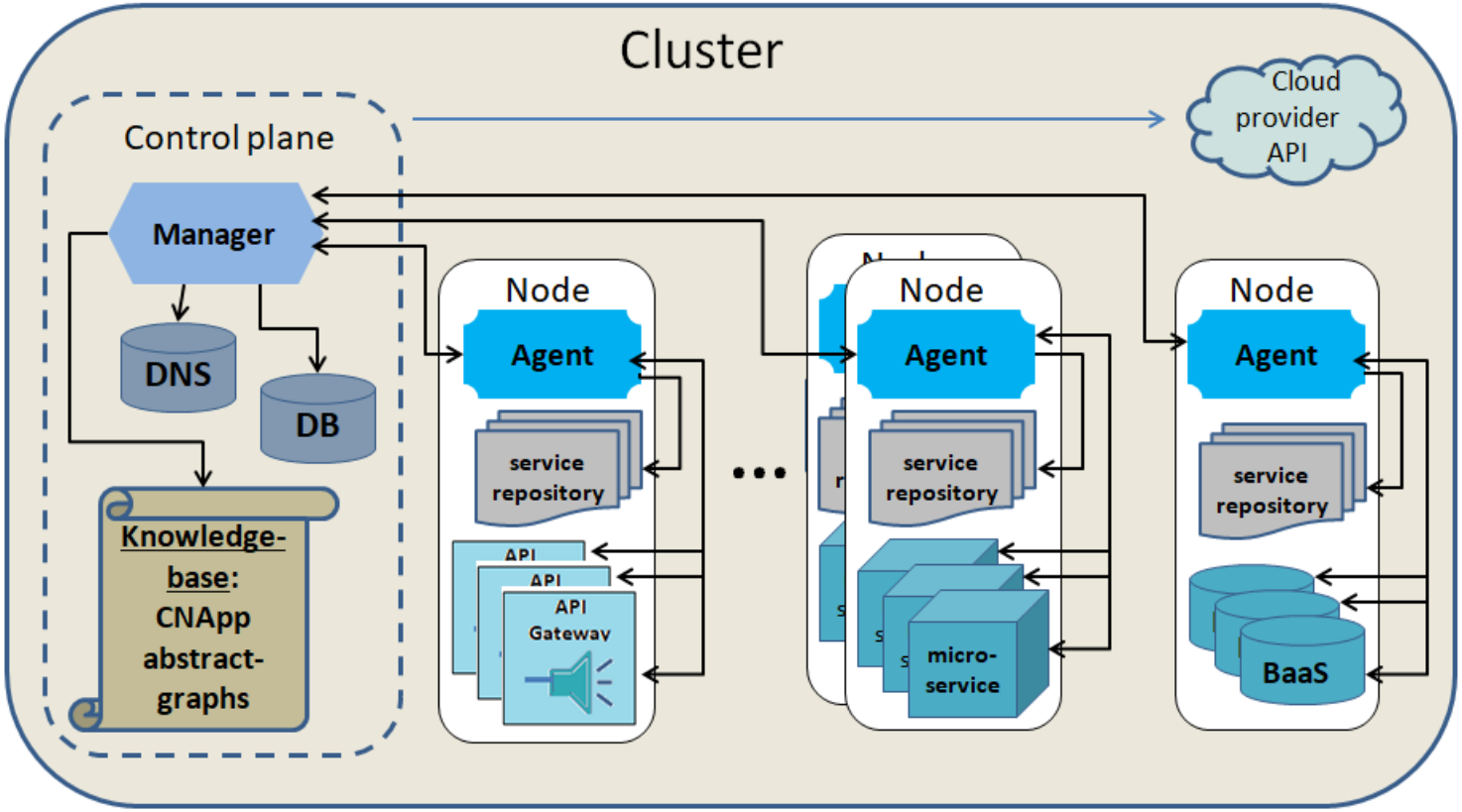
Final remarks

- This is the most complex requirement to be implemented in the codebase of each service participating in SSMMP.

- This requirement can also be seen as a standard recovery mechanism (independent of SSMMP) for handling failures of communication session, e.g. resulting from broken network connections. It seems reasonable to implement these recovery mechanisms in each CNApp service, regardless of SSMMP.

- The rest of the implementation requirements are relatively simple and can be completely separated from the business logic of the services.

The End


Cluster
Control plane
Manager
DNS
DB
Knowledge-base: CNApp abstract-graphs
Cloud provider API
Node
Agent
service repository
API
API
API Gateway
Node
Agent
service repository
micro-service
Node
Agent
service repository
BaaS

Introduction

- Microservices (as a software architecture) were first developed from the service-oriented architecture (SOA) and the concept of Web services (HTTP and WSDL) by Amazon in the early 2000s.
- AWS is short for Amazon Web Services.
- Perhaps Amazon didn’t invent microservices alone.
- AWS became the most successful application of the microservices for Cloud computing at that time.