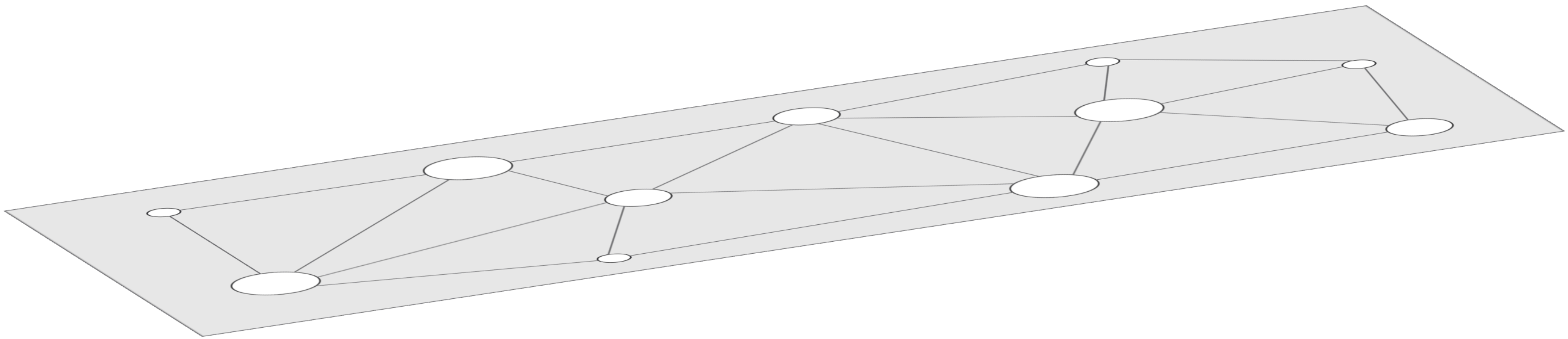
WIND ORCHESTRATION

A system for management and orchestration of distributed heterogeneous cloud

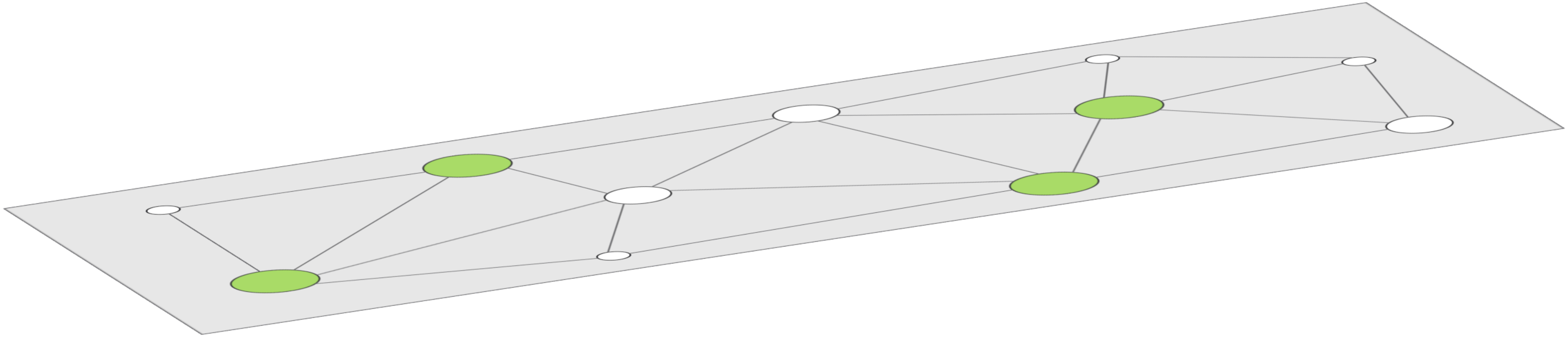
Joacim Halén, Ericsson

Distributed Heterogeneous Cloud



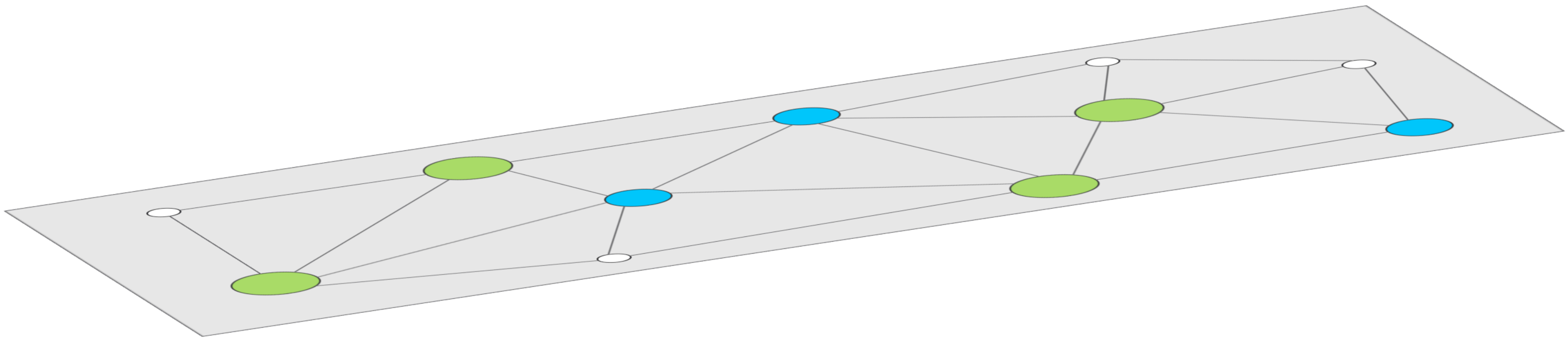
Distributed Heterogeneous Cloud

- Big data center with ~10^5 servers



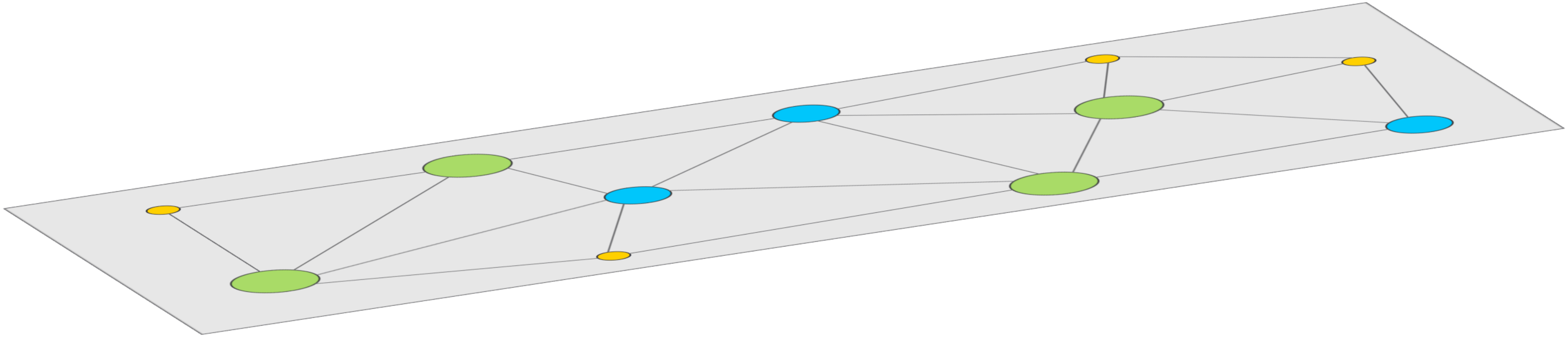
Distributed Heterogeneous Cloud

- Big data center with $\sim 10^5$ servers
- Small data center with $\sim 10^2$ servers

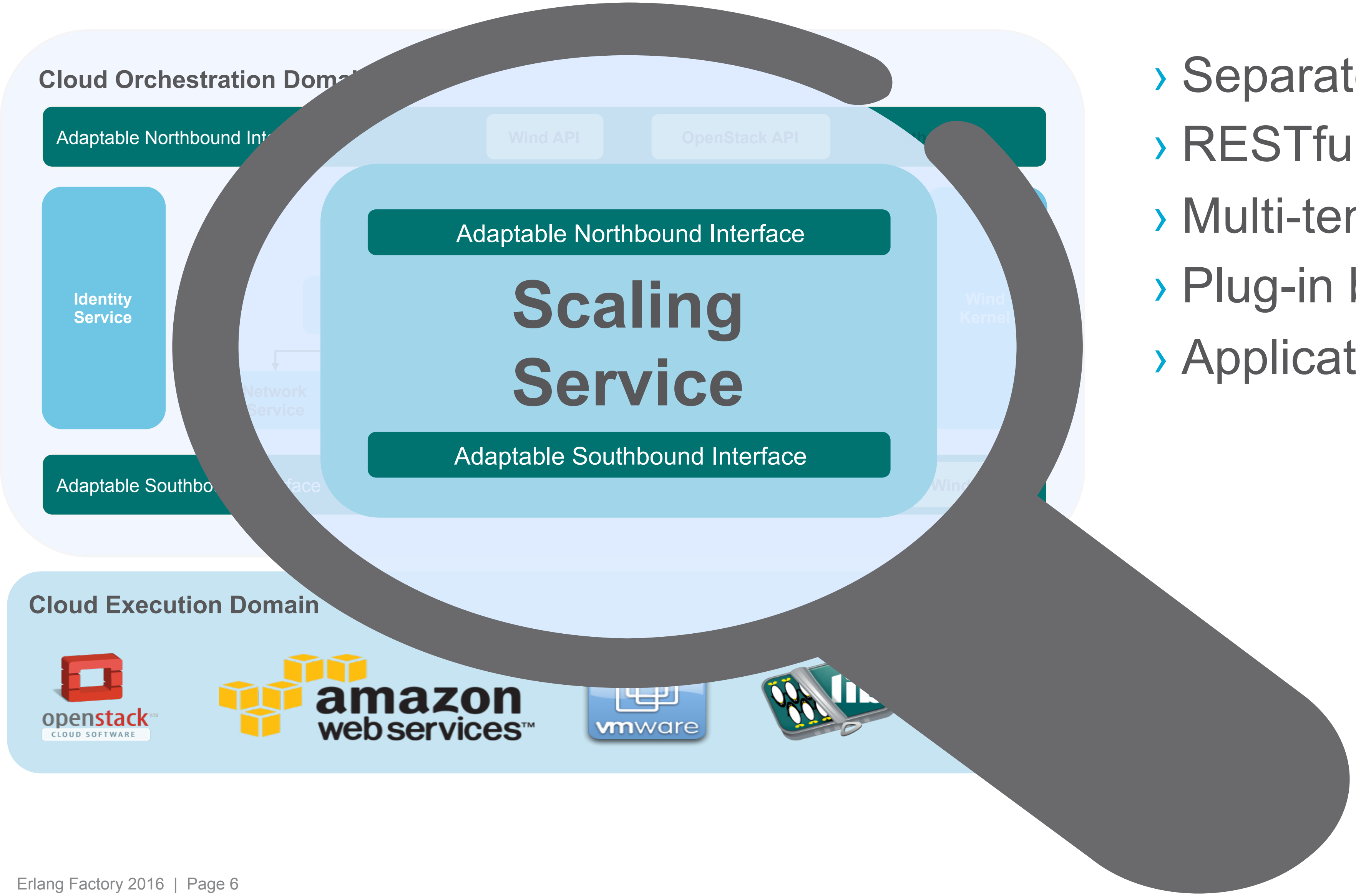


Distributed Heterogeneous Cloud

Each data center may run a different Cloud Operating System or stack, e.g. OpenStack, CloudStack, OpenNebula, etc.



Architecture (simplified)



- Separate services
- RESTful APIs
- Multi-tenant support
- Plug-in based
- Applications can use all APIs

Fundamental Service
Intermediate Level Service
High Level Service

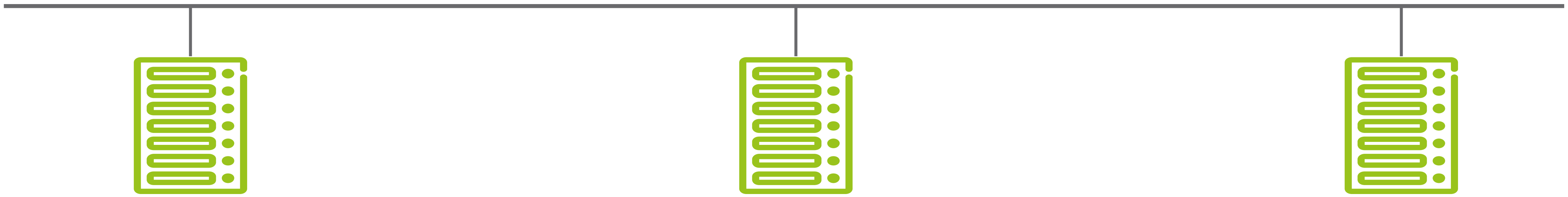
Compute and Network Services

Compute Service

Extended with the Concept of **location**

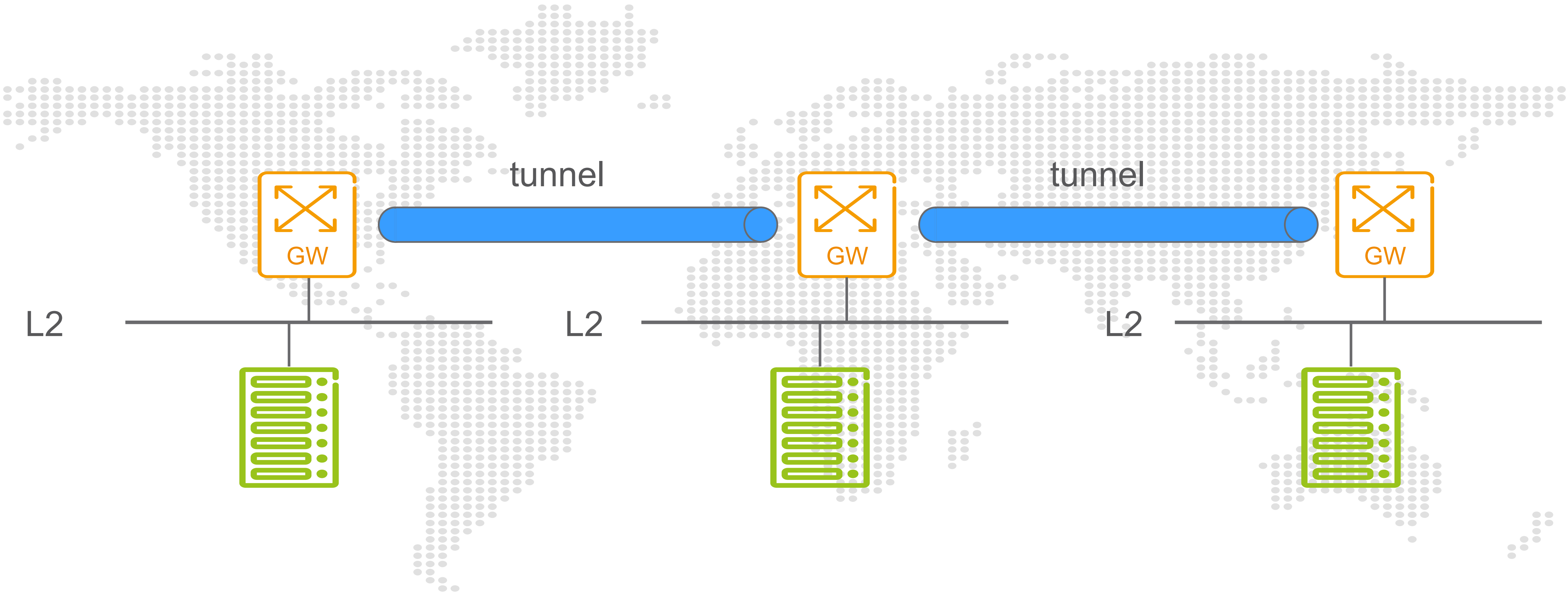
- Other
  - Latency
  - Close to IP
  - Between two nodes
  - At end of longest common path
  - Etc.
- Geographical location
  - Region
  - Country
  - City
  - Data center (node)
    - Rack
    - Host

Simple network



Add context

Possible realization



A Different context

Orchestration Service

Service Container (BNF)

```
BODY         ::= {"service" : {
                     "name" : STRING,
                     "vpcRef" : INTEGER,
                     "parameters" : { PARAMETERS },
                     "definitions" : { DEFINITIONS },
                     "temporals" : [ TEMPORALS ],
                     "scaling" : { SCALING_RULES },
                     "networks" : [ NETWORKS ]} }

DEFINITIONS ::= DEFINITION , DEFINITIONS
              | DEFINITION

DEFINITION  ::= NAME : OBJECT

OBJECT      ::= SERVER | PORT | NETWORK
```

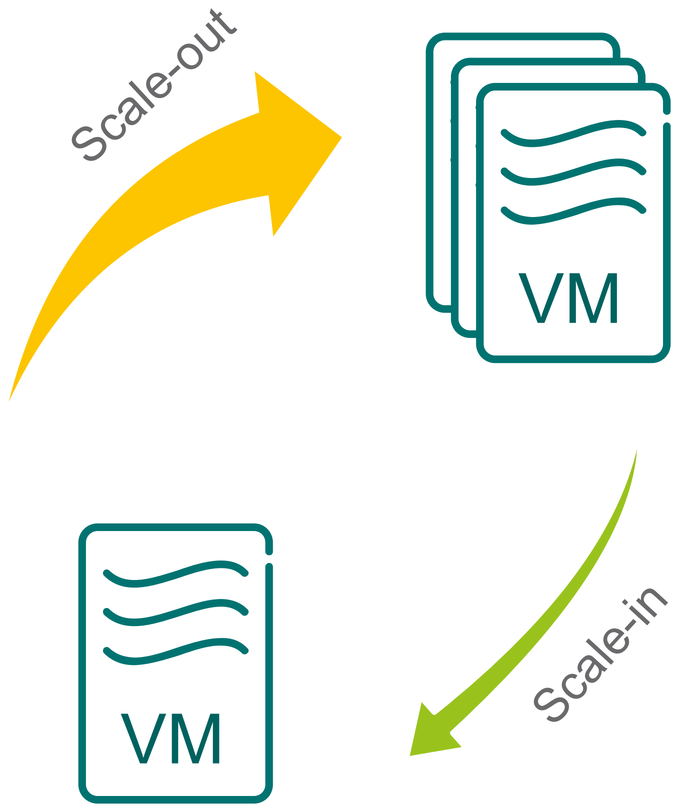
EX1 - specification

```
{
    "service" : {
        "name" : "Example 1",
        "definitions" : {
            "S1" : {"server" : {… "Montreal" …}},
            "S2" : {"server" : {… "San Jose" …}},
            "S3" : {"server" : {… "Stockholm" …}}
        },
        "networks" : [
            {"network" : {
                "layer" : 2,
                "name" : "Example Network",
                "attributes" : {…},
                "ports" : ["S1", "S2", "S3"]}
            }
        ]
    }
}
```

Scaling Service

Scaling Service

- Based on set of application defined rules used as templates for how to add or remove infrastructure resources
- Defines limits on minimal and maximal amount of resources
- Application has full control on how to activate rules:
  - By using API calls
  - By defining automatic triggers specifying metrics to be monitored and thresholds to be met



Scaling Rule (BNF)

```
SCALING_RULE ::= {"scaling-rule" : {
                    ”name” : NAME,
                    ”parameters” : { PARAMETERS },
                    ”initial_parameters” : IPARAMETERS },
                    ”scale_out” : SCALE-OUT,
                    ”scale_in” : SCALE-IN,
                    ”scale_up” : SCALE-UP,
                    ”scale_down” : SCALE-DOWN,
                    ”triggers” : [ TRIGGERS ],
                    ”template” : TEMPLATE,
                    ”notify” : [ RECIPIENTS ]
                 }}
```

A Closer Look

Plug-ins

- Simple “behavior”
- Two callback functions

```
load(Config)   -> {ok, State}
unload(State) -> ok
```

- All user defined functions that are exported must take an extra parameter “State”

```
foo(P1, P2, State) -> {reply, Reply, State}
```

- Plug-ins can be defined to be pre-loaded or loaded at first use
- Plug-ins have a user defined type

PIM – Plug-in Manager

- Basic plug-in management
- Makes sure a plug-in is loaded when needed
- Thread safe, execution of user defined functions in a plug-in is done in the calling process, not in pim
- All calls to a plug-in is done through pim

```
pim:invoke(Name, Function, Args)
```

- Finds plug-in based on name or type
- Search functions to find a plug-in or set of plug-ins
- More complex selection of plug-ins is done in wrappers

Wrappers

- **wpim** – Wind Plug-In Manager
- Location based selection of plug-ins

```
wpim:invoke(Node, Name, Function, Args)
wpim:invoke(Node, Type, Function, Args)
wpim:invoke(NodeA, NodeB, Type, Function, Args)
wpim:invoke(Name, Function, Args)
```

- **drim** – Driver Manager
- Singleton plug-ins, i.e. drivers
- Example, database driver

Code snippet

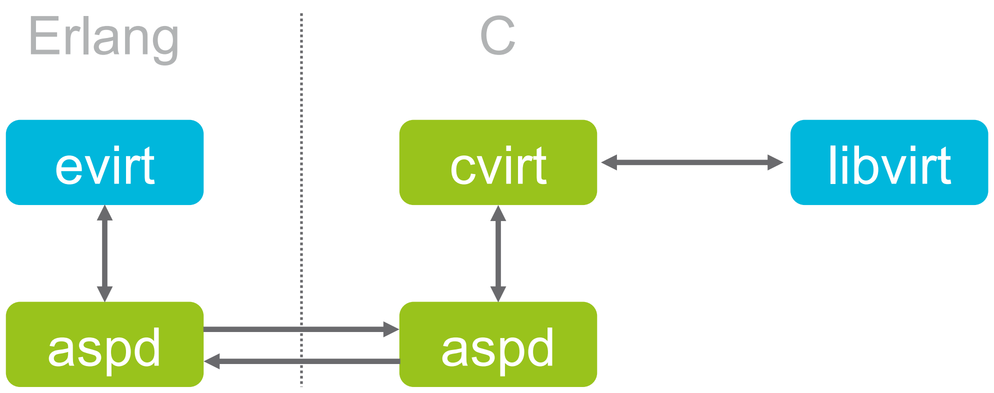
```
…
LocalToken = get_local_token(Tenant, Node),
case wpim:invoke(Node,
                 ?WPIM_COMPUTE,
                 server_create,
                 [Node, LocalToken, Server, Flavor, Image])
of
…
```

Evirt

- Erlang API to libvirt
- One-to-one mapping
- 280+ functions in API
- Supports libvirt 0.9.3
- Full support for callback functions
- Based on aspd

ASPD

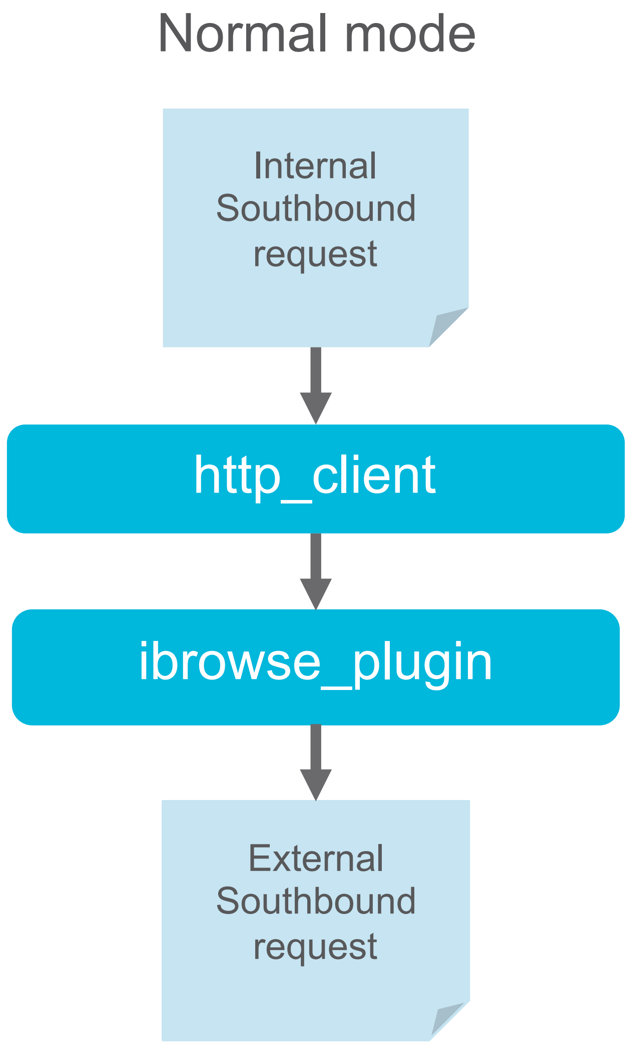
Asynchronous Synchronous Port Driver



- Bridge between libraries
  - Erlang to C
  - C to Erlang
- Simple to use
- Support callback functions
- Library of convenience macros
- Support for logging

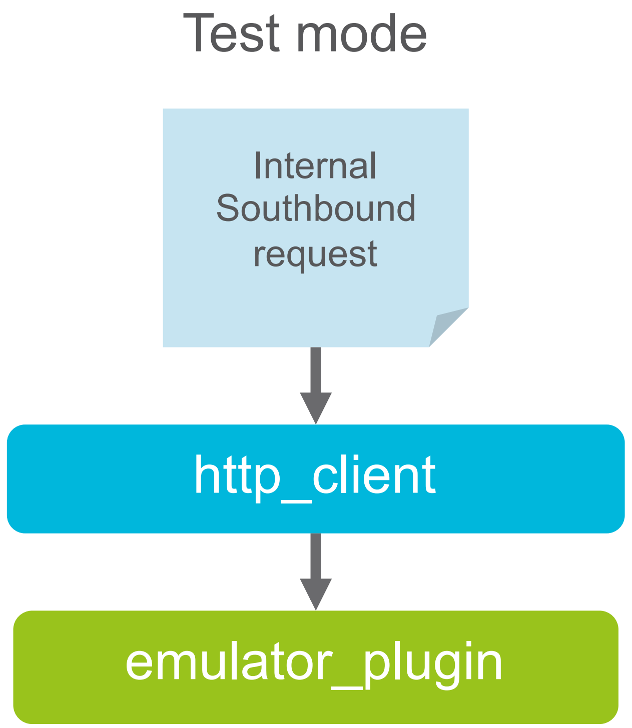
Testing

- Using eunit
- Tests at each level test that level and all levels involved below
- HTTP-client plug-in emulates a distributed OpenStack based cloud
- Wind does not know if it runs against a real cloud or the emulator

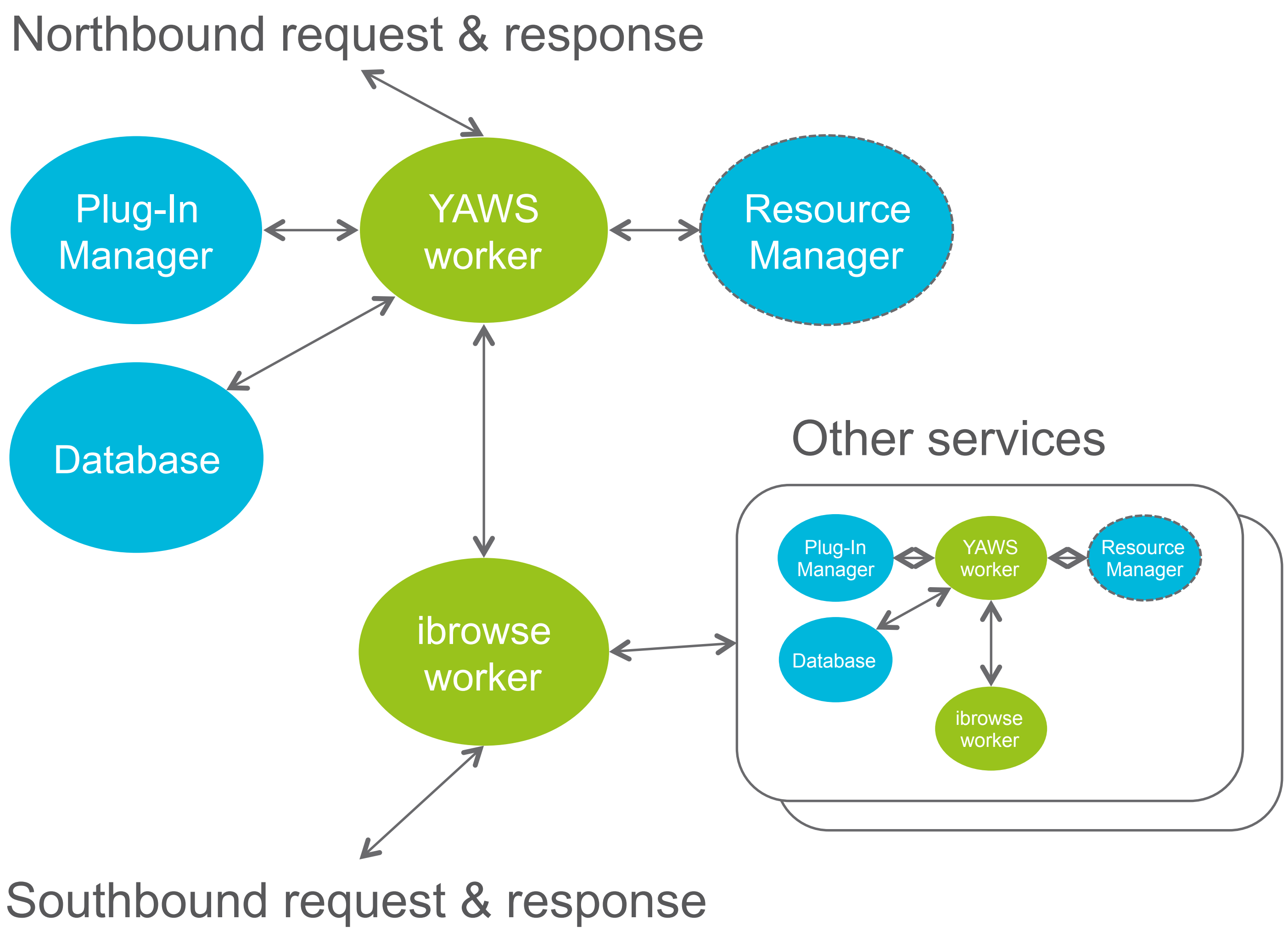


Testing

- Using eunit
- Tests at each level test that level and all levels involved below
- HTTP-client plug-in emulates a distributed OpenStack based cloud
- Wind does not know if it runs against a real cloud or the emulator



Reflection



- Most code handling a request executes in the worker process assigned by YAWS
- Request to internal processes are in most cases very short
- Less risk of deadlock in complicated chains

Artemis

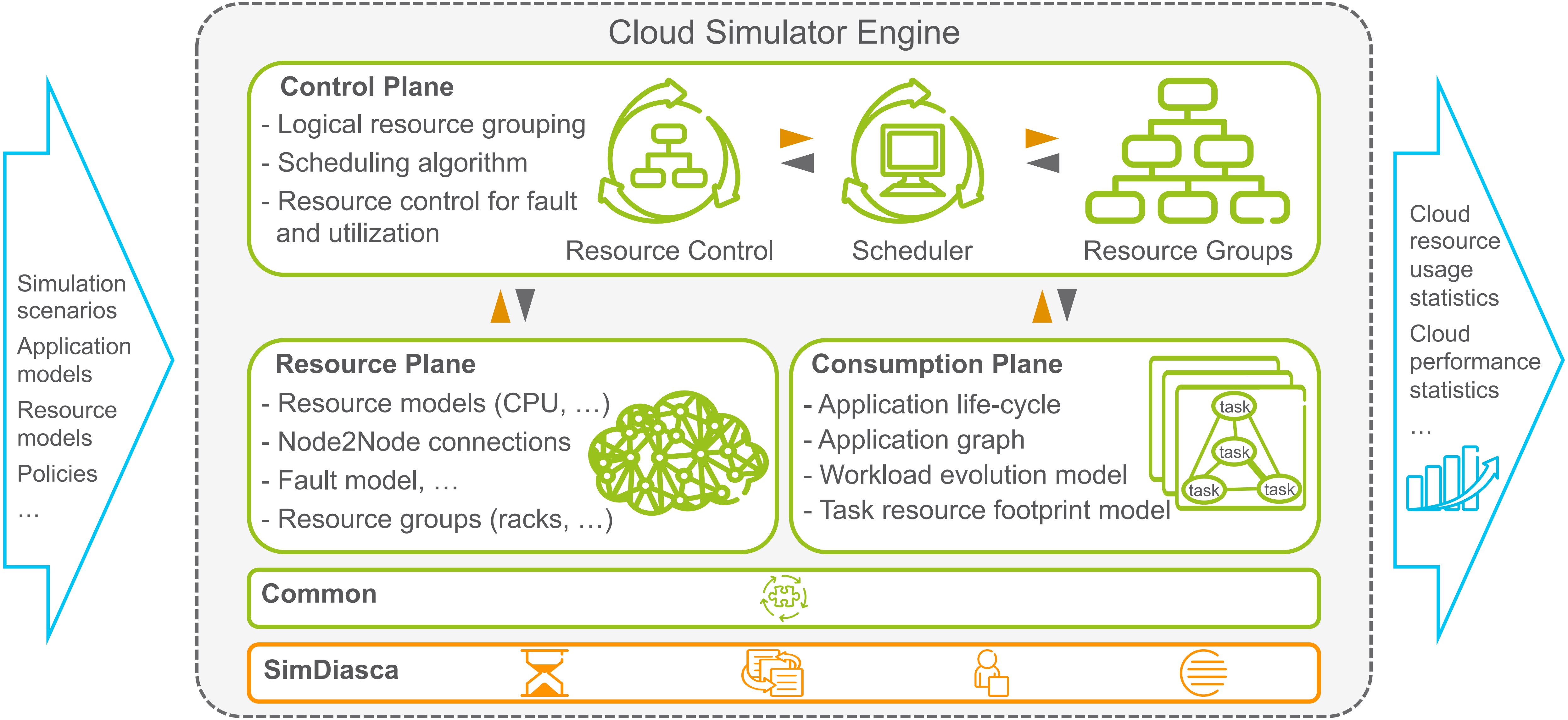
Why ArtEmis?

- Have been focusing on scheduling/placement in very large distributed clouds
  - No large scale physical test-beds
  - Small scale physical test-beds are misleading
  - Thus, simulations!
- Unfortunately, the existing simulation platforms are not suitable for cloud scales
  - Well-known ones only run on a single computer
  - Simulation time does not scale with available resources
  - Thus, they are limited to a few thousand simulation entities and events per second

What is Artemis?

- Artemis is a cloud simulation suite built on top of SimDiasca
- Artemis inherits scalability from Erlang and SimDiasca
  - Simulation run times scale with available resources
  - Handles **millions** of simulation entities and **hundreds of thousands** of events per second
- Provides a set of templates and models for the cloud
- The ultimate goal is to help developers focus on
  - Evolution modelling of both available resources and workloads
  - Development of strategies in as many problem domains within cloud computing as possible
  - No **plumbing!**

Overview



Simulation scenarios
Application models
Resource models
Policies
…
Cloud Simulator Engine
Control Plane
- Logical resource grouping
- Scheduling algorithm
- Resource control for fault and utilization
Resource Control
Scheduler
Resource Groups
Resource Plane
- Resource models (CPU, …)
- Node2Node connections
- Fault model, …
- Resource groups (racks, …)
Consumption Plane
- Application life-cycle
- Application graph
- Workload evolution model
- Task resource footprint model
task
task
task
task
Common
SimDiasca
Cloud resource usage statistics
Cloud performance statistics
…

Example

Declaration of the test module and inclusion of necessary SimDiasca and Artemis libraries

```
-module(generic_control_agent_specialization_stress_test).

-include("test_constructs.hrl").
-include("common.hrl").
-include("resource_plane.hrl").
```

```
-spec run() -> no_return().
run() ->
  ?test_start,
```

Declaration of simulation and deployment settings

```
  SimulationSettings    = #simulation_settings{simulation_name = "Stress Test with Test Agent Inheriting from Generic Control Agent"},
  DeploymentSettings    = #deployment_settings{
    computing_hosts              = {use_host_file_otherwise_local, "sim-diasca-host-candidates.txt"},
    additional_elements_to_deploy = [{".", code}, {"..", code}, {"../../resource-plane", code}, {"../../common", code}],
    enable_data_exchanger        = false,
    enable_performance_tracker   = false
  },
  LoadBalancingSettings = #load_balancing_settings{},
  DeploymentManagerPid  = sim_diasca:init(SimulationSettings, DeploymentSettings, LoadBalancingSettings),
```

Declaration of evolution and physical resource models, and creation of control agents

```
  GIM                   = class_GlobalIdentificationManager:new_link([]),
  Status                = common:create_status(true, {static}),
  Latency_Evolution     = common:create_evolution({distribution, {uni, 100, 1000}}, {constant, 0.1}),
  CPU_Evolution         = common:create_evolution({distribution, {uni, 100, 1000}}, {distribution, {uni, 1, 16}}),
  Memory_Evolution      = common:create_evolution({distribution, {uni, 100, 1000}}, {distribution, {uni, 4, 32}}),
  Disk_Evolution        = common:create_evolution({distribution, {uni, 100, 1000}}, {distribution, {uni, 500, 2000}}),
  Bandwidth_Evolution   = common:create_evolution({distribution, {uni, 100, 1000}}, {distribution, {uni, 100, 1000}}),
  Domain_Evolution      = common:create_evolution({static}, {static}),
  Latency               = common:create_attribute(latency, milliseconds, 0.1, Latency_Evolution),
  CPU                   = resource_plane:create_physical_resource(processing, cores, 16, 0, CPU_Evolution),
  Memory                = resource_plane:create_physical_resource(memory, gigaBytes, 32, 0, Memory_Evolution),
  Disk                  = resource_plane:create_physical_resource(storage, gigaBytes, 2000, 0, Disk_Evolution),
  Bandwidth             = resource_plane:create_physical_resource(network, megabps, 1000, Bandwidth_Evolution),
  Link_1                = resource_plane:create_physical_link(some_connection_point, Status, [Latency], [Bandwidth]),
  Link_2                = resource_plane:create_physical_link(some_connection_point, Status, [Latency], [Bandwidth]),
  Link_3                = resource_plane:create_physical_link(some_connection_point, Status, [Latency], [Bandwidth]),
  Link_4                = resource_plane:create_physical_link(some_connection_point, Status, [Latency], [Bandwidth]),
  Node                  = resource_plane:create_server(GIM, Status, [], [CPU, Memory, Disk], [Link_1, Link_2, Link_3, Link_4]),
  Domain                = resource_plane:create_physical_domain(true, undefined, [Node], Domain_Evolution),
  lists:foreach(
    fun(_) ->
      class_Actor:create_initial_actor(class_GenericControlAgentSpecialization, ["Test Agent 1", [Domain]])
    end, lists:seq(1, 500000)
  ),
```

Running the simulation and finalizing upon termination

```
  SimulationDuration    = 10000,
  DeploymentManagerPid ! {getRootTimeManager, [], self()},
  RootTimeManagerPid = test_receive(),
  RootTimeManagerPid ! {startFor, [SimulationDuration, self()]},
  receive
    simulation_stopped ->
      ?test_info("Simulation stopped spontaneously, specified stop tick must have been reached.")
  end,
  ?test_info("Browsing the report results, if in batch mode."),
  class_ResultManager:browse_reports(),
  sim_diasca:shutdown(),
  ?test_stop.
```

Possible use cases

- Modelling large-scale cloud dynamics
- Methodologies for service placement in very large scale distributed clouds
- Methodologies for dynamic resource management
- Methodologies for fault tolerance, failure resilience and high-availability
- Methodologies for monitoring resource reservation/availability/usage

Q&A

Joacim Halén
Senior Specialist in Software Design and Cloud Automation
joacim.halen@ericsson.com

ERICSSON