IN SUPREMÆ DIGNITATIS

1343

UNIVERSITÀ DI PISA

Dipartimento di Informatica

Orchestration and collocation of application for autonomous drive

Candidato: Tommaso Lencioni
Relatori: Prof. Patrizio Dazzi, Prof. Antonio Brogi

Anno Accademico: 2021/2022

Contents

Abstract This thesis aims to expose advantages of an hybrid architecture that collocate itself between the big computational power but high delay price of Cloud computing and the low delay but small computational power of Edge computing. This is done by simulating a simple leader algorithm and analyzing the results.

Introduction

Chapter 1

Introduction to application allocation problem

The growing interest in the cloud computing paradigm and its application in the automotive led research community to invest in this area (also with european project like H2020 TEACHING).

In this case particular attention is put on the creation of a distributed edge-oriented environment that integrates heterogeneous resources like edge devices, specialized on-site nodes and cloud.

1.1 Application Placement

The application placement problem is a concern for every large scale infrastructure.

When the number of task generated and computational device rise the nearest/strongest offload location is not always the best.

A centralized computation is not an option when the requirement (mainly in terms of latency) are strict.

This leads to the need to have methodologies to find a suitable device for the task.

1.2 State of the art

The state of the art for Cloud Brokering involve static and dynamic analysis of tasks and available nodes in order to establish load distribution prior to

the actual deployment.
Algorithms that accomplish this use mathematical, algorithmic and machine learning techniques.
An overview of those can be taken in this paper from Brogi et al [1].

Chapter 2

Scenario

The scenario considered in this thesis is an intelligent vehicles environment. The way different components interact emulate a real world use although the some parts of the solutions described in the following sections are just for research purposes.

2.1 Vehicles

The users (vehicles) are represented as edge devices with no computational power (more on that in the following section) that generate tasks in an interval of time.
They are mobile to simulate a realistic behavior. This leads to strict delay and mobility requirements that the nodes where the tasks are offloaded must satisfy.
Is with those very constraints that a hybrid model is shown to be effective.

2.1.1 Applications

I analyzed which tasks an actual vehicle would generate.
To maintain the focus on the thesis objective I simplified the complex structure underneath tasks inside the device and decided to represent just the jobs that actually leaves the vehicle.
Doing this way I take for granted all the intra-node processing and communication thus considering every vehicle as an edge device with no computational power. The proposed tasks are the following:

1. `SENSOR_DATA_COMMUNICATION`

   Application that simulates the gathering and communication of data originated by vehicle's sensors.
   This task has a strict limit in terms of delay but the overall computation is light.
   As stated above, we take for granted a preliminary elaboration and anonymization of the data collected by the car done in respect of the privacy limits established with the end user.

2. `ENTERTAINMENT`

   Application that simulates the use of infotainment (music streaming, webradio, etc.)
   This task admits longer delay but is computationally heavier than the previous one.

2.2 Road site units

Road side unit are devices that lay on the border of the road frequently integrated in traffic lights or signs.

Those data centers are simulated with edge data centers.

They represent a first point of arrival of the task sent by the devices.

The communication between the two happens through LAN and undergoes the limitation imposed by both edge device's range and the edge data center's range.
They have computational power and can act as orchestrator.

This means that a device asks the data center to offload the task for it, extending in fact the range on which suitable computational power can be found at the expense of the introduction of communication delay.
They don't generate tasks and they are the core actor in my Leader Algorithm.

Every operation that involves manipulating data coming from the edge devices is simulated by the time taken from the final executor to compute the task returning the result to the source (the edge device).

2.3 Cloud

The Cloud is present as last resort due to high delay introduced in the offload on it.

It is reachable from every device through WAN without the physical limitation of the LAN.

We assume it has an unlimited computational power.

Therefore the only (and huge, in environments of this kind) downside is the time it takes to execute all the tasks queued, making them fail for exceed of delay.

2.4 Thesis's goal

The goal of this thesis is to:

- analyze how the previously described system can be simulated with a simulation software (Pure Edge Simulator),
- how edge data centers organized in communities with a leader through a simple algorithm can represent a valid intermediate solution between pure edge and cloud computation.

Metodology

Chapter 3

Pure Edge Sim

The tool used is **Pure Edge Simulator** (from here after *PES*).

PES is a simulation framework for performance evaluation developed by a group of Tunisian researchers focused on cloud, fog, and pure edge computing environments.

3.1 Requirements

The project is in Java and I used an IDE for my whole work, this simplify the dependencies management.

3.2 Features and limitations

PES is a task oriented simulator.

This trait is inherited from its parent tool Cloud Sim.
Although this approach to simulations is suitable for testing task scheduling on Cloud and Edge, such decoupling represents a major barrier for the implementation of consequentials events involving more than one architecture layer. An insight of the structure can be take in the paper proposed by the authors [2].
For my work I had both to

- extend some classes using a mechanism provided by the authors
- modify core classes that the simulator doesn't plan to be changed.

Every change (mainly workarounds for PES limitations) has been done preserving the working properties of all the project regardless of the type of simulation executed.

3.3 Starting a simulation

Starting from main in MainApplication the method launchSimulation is called.
Configuration files are parsed.
Scenarios (combinations of parameters) are loaded into Iterations. The iterations are executed one by one in a for loop.
Before starting the simulation the models are loaded:

- The data centers are generated (this includes the edge devices).
- The tasks are generated at random for the devices that can generate them (every device have an APPLICATION_LIST).
- An orchestrator is set.
- A network model is set.

The method startInternal of SimulationManager is called.
Before scheduling the task as stated below there is a check for the flag ENABLE_ORCHESTRATORS. If it is false the task's orchestrator is the generating device itself. All the task are scheduled to "this" (the Simulation Manager) with a delay based on the time established during their generation and with the tag SEND_TO_ORCH.
Tasks regarding the simulation (log printing, charts updating and progress bar) are scheduled.
All the classes that can receive a scheduled task (not necessary an edge device's task) implements the method processEvent that evaluate the tag of the event in a switch, trying to match it to a known case.

3.4 Task's life

Every step in a task's life is handled by either Simulation Manager or Network Model. The initial scheduling to itself done for every task as stated in the previous section causes the Simulation Manager to catch the tasks in the

method **process event**.
Due to the SEND_TO_ORCH flag set before the method **sendTaskToOrchestrator** is called. Check if the task is failed. If the orchestrators are enabled the closest device among the non-cloud orchestrators is set as such for the task. Then the task is scheduled immediately with **scheduleNow** to the Network Model with tag SEND_REQUEST_FROM_DEVICE_TO_ORCH.

As stated before the Network Model of the simulation receive the task and in its processEvent method the right case is caught, calling the method **sendRequestFromDeviceToOrch** right away.
If the orchestrator of the task is the generating device itself the task is immediately scheduled to the Simulation Manager with SEND_TASK_FROM_ORCH_TO_DESTINATION tag. Otherwise a new transfer with

- file size = the number of bits of the task's file
- type = REQUEST

is added to **transferProgressList** in order to simulate the request traveling through the network from the source to the orchestrator. Let's consider this case.

Network transfers

The **transferProgressList** is an ArrayList initialized at the creation of the Network Model object.

It is loaded with transfers objects to simulate any kind of displacement of information.

In the method startInternal of Network Model an event with tag UPDATE_PROGRESS is scheduled and repeated every NETWORK_UPDATE_INTERVAL as we can see in processEvent.

Here updateTasksProgress is called every update cycle.

In this method transferProgressList is scanned and, for every transfer in queue, the number of transfers on the same network (both WAN and LAN) and their utilization are evaluated.

- WAN is used when the transfer involves the Cloud.
- LAN is used when two transfers share an endpoint.

Those exchanges undergo the bandwidth limit that is set to the minimum between LAN and WAN. The bandwidth is updated (updateBandwidth) as well as the transfer (updateTransfer).
In this part a crucial role is played by the flag REALISTIC_NETWORK_MODEL. In fact, if true, the remaining file size of the transfer is set after a subtraction between the old remaining file size and the NETWORK_UPDATE_INTERVAL times the current bandwidth. This implies that a simulation done in this way reflects how the network delay is impacting the performance of the whole system.
This is essential to point out because if the flag is false the remaining file size is simply set to 0. Considering the fact that this happens every NETWORK_UPDATE_INTERVAL, if the latter is set to a low value the simulation could be not faithful.
Upon finishing the transfer (therefore the remaining file size is 0) **transferFinished** is called. Here the network usage is updated and the transfer is removed from the queue. According to the type of the transfer (request, task, container, result to orchestrator or result to device) a different "if guard" is satisfied and the corresponding method is called. After that call the energy model is always updated
In our case (REQUEST as type) the method offloadingRequestRecievedByOrchestrator is called.

Here there is a check for the type of task's orchestrator:

- If it's Cloud then the task is scheduled to Simulation Manager with a delay of WAN_PROPAGATION_DELAY and tag SEND_TASK_FRO M_ORCH_TO_DESTINATION
- otherwise same as above except the immediate schedule (no WAN propagation delay if the orchestrator is an Edge Device).

In Simulation Manager the corresponding switch case is caught in processEvent and sendFromOrchToDestination is called. Here there is a check if the task is failed. After that the a suitable VM is searched for the task.

VM Choice

The method **initialize** of SimulationManager's orchestrator is called. The architecture of the simulation is evaluated and the respective method is invoked. Whatever method is chosen the only difference is the array of strings

Architecture (containing the names of the supported architectures in this simulation) passed to the method nesting of assignTaskToVM and findVM.

The latter is up to implementation and utilizes the simulation algorithm to find the suitable VM.
In my case I customized it starting from the proposed Round Robin algorithm aiming to adapt it to my Leader Algorithm orchestration deployment.

assignTaskToVM evaluates the VM (-1 results in a failure for lack of resources) and assign it to the task.

There is a second evaluation for *task.getVm() == Vm.NULL*, this time in SimulationManager.

Here I modified the default behavior adding a second check whether the orchestrator of the task has a leader.

- If it's the case an event containing the task is scheduled to it with tag **TASK_OFFLOAD** and the method returns.
- Otherwise there is a further check whether the orchestrator of the task is also a leader (hence being an isolated leader, more on that later). In this case the same event is scheduled to the orchestrator of the task (being the very leader) and the method returns.

If the *task.getVm() == Vm.NULL* check fails the number of task failed for lack of resources increases and the method returns.
But if the VM is found there is a check:

- If the device that generated the task isn't the same that contains the assigned VM and the orchestrator of the task isn't the data center that contains the assigned VM then a task with tag SEND_REQUEST_FR OM_ORCH_TO_DESTINATION and destination the Network model is scheduled immediately.
- Otherwise the task is scheduled to this (Simulation Model) with tag EXECUTE_TASK.

Let's consider the firts case.

In network model the corresponding case is caught and **sendRequestFromOrchToDest** is called.

A transfer is added to the queue with tag TASK. The transfer behavior is the same as mentioned in the Network transfer subsection.

Upon finishing the transfer the corresponding if is caught and executeTaskO-rDownloadContainer is called.

Here there is a check for the ENABLE_REGISTRY, in that case a new task with tag DOWNLOAD_CONTAINER is scheduled.

Otherwise the task is scheduled to the Simulation Manager according to the type of data center of the VM assigned to the task (schedule with WAN_PROPAGATION_DELAY for Cloud and schedule immediately for the Edge) with the tag EXECUTE_TASK.

In the simulation manager the corresponding branch is caught. There is a check for failure, then submitCloudlet is called on the broker (created at the initialization). The CPU utilization of the VM is increased accordingly and the energy model too.

The task is executed by the Cloud Sim part of the project.

At its termination (CLOUDLET_RETURN case in processEvent of the DatacenterBrokerSimple class extension) the task is immediately scheduled to the simulation manager with tag TRANSFER_RESULTS_TO_ORCH.

In Simulation Manager TRANSFER_RESULTS_TO_ORCH case is taken. The CPU percentage is remove and the method sendResultsToOchestrator is called.

There is a check if the task is failed. Then:

- If the source device isn't equal to the data center that contains the associated VM the task is scheduled immediately to the network model with tag SEND_RESULT_TO_ORCH.
- otherwise the task is immediately scheduled to this (Simulation Manager) with the tag RESULT_RETURN_FINISHED.

Let's consider the first case. In network model the corresponding case is caught and the method sendResultFromDevToOrch is called. Here there is a check:

- if the orchestrator of the task isn't the device that generated the task then a file transfer is started with tag RESULTS_TO_ORCH
- otherwise the task is immediately scheduled to this (Network Model) with the tag SEND_RESULT_FROM_ORCH_TO_DEV)

Let's consider the first case.

Upon finishing the transfer returnResultToDevice is called. A check whether the orchestrator or the data center of the VM assigned to the task are Cloud. In that case the task with tag SEND_RESULT_FROM_ORCH_TO_DEV is scheduled to this (Network Model) with WAN_PROPAGATION_DELAY.

Otherwise the same is done without delay.

The corresponding case is caught and the method sendResultFromOrchToDev is called. Here a new file transfer is added with tag RESULTS_TO_DEV.

Upon finishing the transfer the last else is caught and the method resultsReturnedToDevice is called. This schedules immediately the task for the Simulation Model with tag RESULT_RETURN_FINISHED.
In Simulation Model the corresponding case i caught. A failure check is done, then a customizable method of the simulation orchestrator is called (resultsReturned) and the number of task is increased.

3.5 Leader Algorithm

Idea An edge data center should be able to elect a leader among the other edge data centers in reach if they meet some arbitrary conditions.

A leader (which can't have a leader) receives tasks from edge data centers in its community when they can't execute them and tries to offload them to other nodes in its community.

As last chance, hence when there are no data center left in the community and still no suitable VM is found, the task is offloaded on the Cloud.

This implies that both Edge and Cloud must be present in computation paradigm in order to accomplish this.

Implementation

LeaderEdgeDevice class

The custom class *LeaderEdgeDevice* extends DefaultDataCenter.

Community discovery In *startInternal* a task with custom tag COMMUNITY_DISCOVERY is scheduled with INITIALIZATION_TIME + 1 delay.

In *processEvent* the tag is caught.

There is a check for:

- the device being an edge data center,
- the device being an orchestrator,
- "LEADER" being the orchestrator deployment method.

This allows that only orchestrator data centers, if the simulation uses leaders, to have/be a leader and being part of a community.

If everything is true then the method **discover** is called. This procedure allows every edge data center that takes part in the leader algorithm to have its own community of neighbor nodes.

- in a loop every datacenter is taken into account and is checked whether the candidate:
  - is not the same data center as the one evaluating,
  - is an edge data center
  - it is an orchestrator
  - is in data center range with the one evaluating.

  If every condition is met the candidate is added to a personal list (community).
- At the end of the loop the community is sorted in descending order according to the data center MIPS.

Leader settlement After INITIALIZATION_TIME + 10 delay an event with tag LEADER_SETTLE is scheduled.

In *processEvent* the tag is caught. There is the same check as COMMUNITY_DISCOVERY, then the **settle** method is called.

- A data center should lead if:
  - its community is empty.

    In this case the edge data center is isolated therefore is leader of its own.
  - its MIPS are greater of the one of the first data center in its community (thus being the strongest among the one it have in range).

- If the condition above aren't met there is a check whether one of the node in its community has it as first element (thus possible leader). Also in this case the node should lead.
- If a node should lead it's removed from the communities that don't have it as first data center in terms of MIPS.
- Then if a node should lead its community is rebuild with nodes that are weaker and have it as first element in their community.

  The method terminates with a further sorting of the newly build community.

Leader confirmation After INITIALIZATION_TIME + 20 delay an event with tag LEADER_CONFIRMATION is scheduled.

In *processEvent* the tag is caught. There is the same check as the two branches above, then the **confirmation** method is called. Upon ending the loop if a leader has been found the current data center is added to its subordinates list and the flag isLeader is set to true (in case it has a leader as well).

Otherwise the orchestrator flag is removed as well as the data center presence in Orchestrators List.

I choose a static criterion but it's customizable and can be dynamic. It's sufficient to reschedule the events

Leader algorithm

Simulation Manager In Simulation Manager I modified the default behavior of the method sendFromOrchToDestination, hence where is decided which VM will execute the task.

This class isn't customizable according to the simulator so I had to modify it directly.
I introduced 2 negative integer "error code" beside -1 (the default error integer returned when no VM has been found).

- -2: the task must be sent to the leader
- -3: the task must be sent to the cloud

Those codes are returned from the method my_initialize.

A switch catch the correct corresponding case scheduling SEND_TASK_FROM_ORCH_TO_DESTINATION adding a delay in case of scheduling to Cloud.

This works by changing the orchestrator of the task in **my_initialize**. Although not correct by a practical point of view this is the only way I came up for including the information about the next datacenter to offload on without modifying the task.
Due to the low flexibility of the tool I had also to specify the reason of failure distinguishing from failing for lack of resources and simply rescheduling the task.

LeaderEdgeOrchestrator In order to find a suitable VM using this hierarchy I created a simulation algorithm called LEADER.

I extended Orchestrator in order to implement a custom findVM method.

This implies to implement my_initialize and the methods corresponding to the various architectures.

my_findVM is called only in edgeAndCloud method because in other architectures there are less or more devices than needed to work.

If LEADER is used as simulation algorithm there is a check for the presence of both Cloud and Edge in the simulation architecture. If it is not the case an error is displayed and the simulation stops.

Otherwise the architecture is restricted to "Edge" and the method **leader** is called.

The algorithm simply iterates through hosts and VMs of the phase's devices in search for a VM that meet the condition imposed by the code.

The condition based on which a VM is selected is completely arbitrary.

I used one based on task's length, MIBS and latency.

This is the order of seek for a suitable VM:

1. Phase 0: among the hosts of the orchestrator.
2. Phase 1: among the hosts of the leader of the original orchestrator and its subordinates.

There should not be a situation where the orchestrator has no leader and is chosen for the previous step so the presence of a leader should be certain at this phase.

If the previous steps should result in no VM selected a try is done in phase 2 with the algorithm INCREASE_LIFETIME (could have been any other algorithm) on the Cloud.

Phases Are determined by the type of the orchestrator. Due to the fact that it changes in every step it's easy to tell in which phase the algorithm is in. This is an escamotage used to bypass PES limitation in flow control of the tasks.

3.5.1 Notes and assumptions

Static election condition The leader election is done once at the start of the simulation because the conditions on which a device is elected as leader are statics.

This can be changed rescheduling the election with a delay in case of dynamics conditions.

Leader's omniscience In phase 1 there is a check for every host of every subordinate of the leader.

This implies an omniscience by the latter.

This is not realistic because the method findVM could discover a VM not owned by the orchestrator neither reliable to do in PES due to the continuous allocation of VM for tasks so this implementation is purely theoretical.

Chaining A case of chaining can occur when the data center A has B as leader but B in turn has C as leader.

Both A and B are orchestrator so they can both receive tasks.

During findVM, if a task has A as orchestrator and it doesn't meet the conditions, the task is initially offloaded to B and its subordinates.

If, again, no VM are found the task goes to C and its subordinates.

The cloud, as always, is the last resort.

Recursion A particular case of the situation above is when A = C.

This is a problem because there is no way to keep track of the devices that already checked for condition on a task.

This could results in a endless loop. (TOFIX)

Isolation If an edge data center orchestrator is isolated in such manner that it doesn't have both leader and subordinates it became useless.

Not having a leader elect it as such but it has no subordinates so it won't receive tasks due to the fact that is removed from the orchestrators list.

Parity In case of condition parity between data centers every device will act as leader, thus not being an orchestrator neither having subordinates.

This could lead to an empty orchestrators list.

Rejection In this implementation data center can't reject to be elected as leader.

This will force it to execute tasks of its self proclaimed subordinates.

This behavior is easily fixable by placing a check for the value of a custom setting flag (”consent”) before electing.

Chapter 4

Simulations and results

TODO LIMITAZIONE ORCHESTRAZIONE

Chapter 5

Conclusion

Bibliography

[1] Antonio Brogi et al. "How to place your apps in the fog: State of the art and open challenges". In: *Softw. Pract. Exp.* 50.5 (2020), pp. 719–740. DOI: `10.1002/spe.2766`. URL: `https://doi.org/10.1002/spe.2766`.

[2] Charafeddine Mechalikh, Hajer Taktak, and Faouzi Moussa. "PureEdgeSim: A simulation framework for performance evaluation of cloud, edge and mist computing environments". In: *Comput. Sci. Inf. Syst.* 18.1 (2021), pp. 43–66. DOI: `10.2298/CSIS200301042M`. URL: `https://doi.org/10.2298/CSIS200301042M`.