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 tries to...

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 Cloud Brokering

The cloud brokering

1.2 State of the art

The state of the art

Chapter 2

Scenario

The scenario considered in this thesis is an intelligent vehicles environment. The way different components interact reflects and emulate a real world use.

2.1 Vehicles

The users (vehicles) are represented as edge devices with no computational power that generate tasks in an interval of time.
They are mobile to simulate a realistic e behavior, this leads to strict delay and mobility requirements.

2.1.1 Applications

The proposed tasks are the following:

1. `SENSOR_DATA_COMMUNICATION`

   Application that simulates the gathering and communication of data originated by vehicle's sensors.
   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.)

2.2 Road site units

Road side unit are devices that lay on the border of the road (possibly integrated in traffic light/sign).

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 devices'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 some communication delay.

They don't generate tasks and they are the core actor in my Leader Algorithm.

2.3 Cloud

The Cloud is present as last resort due to high delay introduced in the offload on it. It offer computational power and is reachable from every device through WAN without the physical limitation of the LAN.

2.4 Thesis's goal

The goal of this thesis is to analyze how the previously described system can be simulated with a software and to propose an algorithm that makes the edge data centers follow a hierarchy model.

Metodology

Chapter 3

Pure Edge Sim

The tool used is **Pure Edge Simulator**.

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 but nullify the possibilities of scripting the tests.

3.2 Features and limitations

Pure Edge Sim (*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 [LINK BIBLIOGRAFIA].

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 the Simulation Manager's orchestrator is called. The architecture of the simulation is evaluated and the respective method is called. Whatever method is called 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 in order to implement the Leader Algorithm.

I used negative integers returned from my_initialize in order to distinguish the step of leader offload.

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 the VM=Null (this time in the Simulation Manager) then 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_FROM_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 executeTaskOrDownloadContainer 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 (that can have a leader as well) can execute tasks on its VM or offload them to its subordinates.

As last chance the task is offloaded on the Cloud.

Implementation

LeaderEdgeDevice class

The custom class (*LeaderEdgeDevice*) extends DefaultDataCenter.

In *startInternal* a task with tag LEADER_ELECTION is scheduled with INITIALIZATION_TIME + 1 delay.

In *processEvent* the custom tag is caught.

There is a check for:

- the device being an edge data center,
- the device being an orchestrator.

  This allows data centers not flagged as orchestrators to not have a leader (although possibly being elected as such by other data centers, even that case is easily fixable).
- "LEADER" is the orchestration method,

If everything is true then the method **leader** is called.

- in a loop every datacenter is taken into account and is checked whether it:
  - is not the same data center as the one evaluating,
  - is an edge data center
  - the distance between the two data centers is smaller than the range of the edge data centers.
- If it is the case then there is an evaluation about the MIPS (I used that criterion for electing a leader).

– If the MIPS of the candidate are greater than ones of the evaluator
– and the max MIPS of the data center seen until that point is lower than the MIPS of the candidate

then the candidate becomes the (potentially temporary) leader, the max MIPS seen is set to the new leader's MIPS.

Upon ending the loop if a leader has been found the current datacenter 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.

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.

I introduced 2 negative integer "error code" beside -1 (no VM 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.
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 architecture there are less or more devices involved.

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 stopped.

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

Here this order of seek for a suitable VM is followed:

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 anything else) on the Cloud.

The condition based on which every level is scanned in search for a VM is completely arbitrary.

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.

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.

3.5.1 Notes

Static election condition The leader election is done once at the start of the simulation because the conditions on which a device is elected 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. (TODO)

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.

My actual implementation take as granted the omniscience of the edge data center in regards of its leader and its subordinates.

Chapter 4

Simulations and results

Chapter 5

Conclusion