Tutorial Mobility Manager (LTE)

Jump to bottom

Roberto Riggio edited this page on Aug 18, 2019 · 1 revision

Table of Contents

Objective

In this section we shall illustrate how to implement a simple handover management application for LTE networks. This application is required to:

- Periodically check if one or more VBS is overloaded, i.e. the PRB utilisation in either the downlink or the uplink direction is above a predefined threshold;
- Handover UE or more UE from an overloaded VBS to another VBS whose PRB utilisation is below a certain threshold.

The complete implementation of the Handover Manager Network application discussed in this tutorial can be found here.

Pre-requisites

In order to run this Network application the following requirements have to be met:

- A Slice with at least two VBSes must be created (tutorial).
- The HandoverManager module has been loaded (tutorial)

Launch method & Initializations Tasks

The Handover Manager class definition and initialization methods are reported below:

```
class HandoverManager(EmpowerApp):
    def __init__(self, **kwargs):
        self.__load_balance = True
        self.__s_dl_thr = 10
        self.__s_ul_thr = 10
        self.__t_dl_thr = 30
        self.__t_ul_thr = 30
        self.__rsrq_thr = -20
        self.__max_ho_from = 1
        self.__max_ho_to = 1
        EmpowerApp.__init__(self, **kwargs)
        self.vbsup(callback=self.vbs_up_callback)
        self.uejoin(callback=self.ue_join_callback)
```

As it can be seen, the initialization tasks registers two triggers: *uejoin* and *vbsup*.

The launch method for the Handover Manager application is shown below:

```
def launch(tenant_id,
           every=DEFAULT_PERIOD,
           load_balance=True,
           s_dl_thr=30,
           s_ul_thr=30,
           t_dl_thr=30,
           t_ul_thr=30,
           rsrq_thr=-20,
           max_ho_from=1,
           max_ho_to=1):

    return HandoverManager(tenant_id=tenant_id,
                           every=every,
                           load_balance=load_balance,
                           s_dl_thr=s_dl_thr,
                           s_ul_thr=s_ul_thr,
                           t_dl_thr=t_dl_thr,
                           t_ul_thr=t_ul_thr,
                           rsrq_thr=rsrq_thr,
                           max_ho_from=max_ho_from,
                           max_ho_to=max_ho_to)
```

As it can be seen the *launch* method has several parameters: the mandatory *tenant_id* parameter, the various threshold used by the application, and the *every* parameter specifying the control task period.

Handover

The *vbsup* trigger is used in order to execute the *mac_reports* primitive when a VBS connects to the controller. This is needed in order to update the slice's network view in terms of PRB utilization.

The *vbsup* trigger callbacks method is reported below:

```
def vbs_up_callback(self, vbs):
    for cell in vbs.cells:
        report = self.mac_reports(cell=cell,
                                  deadline=self.every,
                                  callback=self.mac_reports_callback)
```

The *uejoin* trigger is used to executed the *rrc_measurements* when a UE joins the slice. This is needed in order to update the slice's network view in terms of RSRP/RSRQ quality.

The *uejoin* callback method is reported below:

```
def ue_join_callback(self, ue):
    measurements = \
        [{"earfcn": ue.cell.DL_earfcn,
          "interval": 2000,
          "max_cells": 2,
          "max_meas": 2}]
    self.rrc_measurements(ue=ue, measurements=measurements,
                          callback=self.rrc_measurements_callback)
```

The *loop* method in the *HandoverManager* class is called periodically with period defined by the *every* parameter (in ms).

```
def loop(self):
    """ Periodic job. """

    if not self.load_balance:
        return

    self.log.info("Running load balancing algorithm...")

    # Dictionary containing VBS which qualifies for performing handover
    ho_from_vbses = {}

    # Dictionary containing VBS which qualifies to receive handed over UEs
    ho_to_vbses = {}

    # Check the condition cellUtilDL > dl_thr or cellUtilUL > ul_thr.
    for vbs in self.vbses():

        if not vbs.is_online():
            continue

        for cell in vbs.cells:

            if not cell.mac_reports:
                continue

            self.log.info("Cell %s: DL: %f / %u / %u", cell,
                          cell.mac_reports['DL_util_last'], self.s_dl_thr,
                          self.t_dl_thr)

            self.log.info("Cell %s: UL: %f / %u / %u", cell,
                          cell.mac_reports['UL_util_last'], self.s_ul_thr,
                          self.t_ul_thr)

            if cell.mac_reports['DL_util_last'] > self.s_dl_thr or \
               cell.mac_reports['UL_util_last'] > self.s_ul_thr:

                if vbs.addr not in ho_from_vbses:
                    ho_from_vbses[vbs.addr] = {"vbs": vbs,
                                               "cells": [],
                                               "ues": self.ues(vbs),
                                               "max_ho_from": 0}

                ho_from_vbses[vbs.addr]["cells"].append(cell)

            if cell.mac_reports['DL_util_last'] < self.t_dl_thr or \
               cell.mac_reports['UL_util_last'] < self.t_ul_thr:

                if vbs.addr not in ho_to_vbses:
                    ho_to_vbses[vbs.addr] = {"vbs": vbs,
                                             "cells": [],
                                             "ues": self.ues(vbs),
                                             "max_ho_to": 0}

                ho_to_vbses[vbs.addr]["cells"].append(cell)

    # Find UEs and neighboring cells satisfying the handover conditions
    ho_info = {}

    for svbs in ho_from_vbses:

        # already too many hos from this vbs
        if ho_from_vbses[svbs]["max_ho_from"] >= self.max_ho_from:
            continue

        for ue in ho_from_vbses[svbs]["ues"]:

            for tvbs in ho_to_vbses:

                # too many ho to this vbs
                if ho_to_vbses[tvbs]["max_ho_to"] >= self.max_ho_to:
                    continue

                # pick best cell in vbs
                for cell in ho_to_vbses[tvbs]["cells"]:

                    # ignore current cell
                    if cell == ue.cell:
                        continue

                    # pick cell from measurements
                    if cell not in ue.rrc_measurements:
                        continue

                    current = ue.rrc_measurements[ue.cell]["rsrq"]
                    new = ue.rrc_measurements[cell]["rsrq"]
            if new > current and new > self.rsrq_thr:
                ho_info[ue.ue_id] = {"ue": ue, "cell": cell}

        if ue.ue_id in ho_info and ho_info[ue.ue_id]:
            tvbs = ho_info[ue.ue_id]['cell'].vbs.addr
            ho_to_vbses[tvbs]["max_ho_to"] += 1
            ho_from_vbses[svbs]["max_ho_from"] += 1

    # Handover the UEs
    for handover in ho_info.values():
        ue = handover['ue']
        cell = handover['cell']
        self.log.info("%s -> %s", ue, cell)
        ue.cell = cell
```

▸ Pages 25

Getting Started

- Introduction
- Terminology
- Network Setup
- Setting up the WTP
- Setting up the CPP
- Setting up the VBS
- Setting up the EmPOWER Controller
- Setting up the Backhaul Controller

Using EmPOWER

- Publications

Intent Based Networking

- Introduction

Downloads

- Pre-built WTP Firmwares

Developers

- REST API documentation
- Python API documentation
- Python API (WiFi/LVAP)

- Python API (LTE)
- Python API (Click/LVNF)

Tutorials

- Mobility Manager (WiFi)
- Mobility Manager (LTE)
- Service Function Chaining

Support

- Mailing List

Acknowledgements

- Acknowledgements

Clone this wiki locally

```
https://github.com/clicknf/clicknf.github.io.wiki.git
```