HetNetSim

Generated by Doxygen 1.8.6

Mon Oct 10 2016 19:17:00

Contents

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 AP Class Reference

```
#include <AP.h>
```

Public Member Functions

- AP ()

  METHODS.
- void AP_init (int ID, double coordinates[2], parameters params)
- void ADD_STA (void ∗obj)
- void jump_time (int jump)
- int check_buffer ()
- void sched_tx ()
- void check_tx_time ()
- void callback_tx (void ∗obj, std::string type)

  call backs
- void callback_fintx (void ∗obj, std::string type)
- void add_listener (void ∗obj, std::string type)

  framework support functions
- void notify_listener (int EVENT)

Public Attributes

- int ID
- std::string TYPE = "AP"

  Device Type.
- double height

  AP height from the ground.
- double coordinates [2]

  AP coordintes [x,y].
- std::vector< int > MCS_DL

  Vector for the DL MCS corresponding each STA.
- int MAX_MCS

  Total MCS.
- int CURRENT_STA_DL = 0

  Current STA being served in DL.

- int CURRENT_STA_UL

  Current STA doing UL.
- std::vector< double > ACTIVE_Devices

  Vector containing the distance of all the Active devices int the network.
- std::vector< double > ACTIVE_Devices_power

  Vector containing the tx power of all the Active devices int the network.
- std::vector< long long > ALLOC_data_DL
- std::vector< long long > ALLOC_data_UL
- bool TX_ind = 0
- bool RX_ind = 0
- double Pt_W

  Transmit power of the AP in Watts measured over 1 subcarrier.
- std::vector< std::list< long long > > Buffer_DL

  vector of list of files per user: mimicks the packet que
- std::vector< long long > SUM_BUFF_DL

  vector containing the total data per STA yet to be tranmistted in DL
- long long BUF_Ind = 0
- long long TOT_BUFF = 0

  Total data yet to be transmitted.
- long long Channel_occupy_time = 0

  channel occupy time for transmission
- std::vector< std::list< long long > > Time_file_arrived_DL

  vector of list of time a file arrived per user: complements Buffer DL
- std::vector< long long > TTTDeliverOFile_DL

  vector containing Time in micro seconds took to delive a file in DL used for STAISTICS
- std::vector< double > SINR

  vector containing SINR in dB in DL for each transmission used for STAISTICS
- long long total_files = 0

  total files received used for STATISTICS
- long long served_data = 0

  Total served data in DL used for STATISTICS.
- long long pumped_data = 0

  Total data pumped into the channel in DL used for STATISTICS.
- std::vector< long long > attempt

  Total attempts : Used for MCS adaptation.
- std::vector< long long > success

  Total success : Used for MCS adaptation.
- int CW

  current contention window size
- std::vector< void * > listener_STA
- std::vector< void * > listener_AP

  vector containing the access point object handles

3.1.1 Detailed Description

Class AP IEEE 802.11ac Implements 802.11 DCF PHY is completely Abstracted Access Point node contains a traffic generator + lower MAC + PHY abstraction

Definition at line 28 of file AP.h.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 AP::AP ( )

METHODS.

Constructor

Definition at line 30 of file AP.cpp.

```
30         {
31 }
```

3.1.3 Member Function Documentation

3.1.3.1 void AP::add_listener (void * *obj*, std::string *type*)

framework support functions

ADD listener type station This function adds a STA/AP to the listeners group INPUT: object handle (obj) , object type (obj.TYPE)

Definition at line 758 of file AP.cpp.

```
758                                               {
759     // if the listener is AP
760     if (type == "AP")
761         // ADD to AP group
762         listener_AP.push_back(obj);
763     // if the listener is STA
764     if (type == "STA")
765         // ADD to STA group
766         listener_STA.push_back(obj);
767 }
```

3.1.3.2 void AP::ADD_STA (void * *obj*)

Associate STA to the AP Adds the STA object handle to the vector BS_UEs

Definition at line 117 of file AP.cpp.

```
117                     {
118     // Push STA handle into the vector
119     BS_UEs.push_back(obj);
120 }
```

3.1.3.3 void AP::AP_init (int *ID_in*, double *coordinates_in[2]*, parameters *params*)

initilaizer function Initializes the object after its creation INPUT: ID , coordinates, parameters Data allocated in downlink for each STA This is necessary as there is no explicit signalling of control information

Data allocated in uplink from each STA This is necessary as there is no explicit signalling of control information

Definition at line 37 of file AP.cpp.

```
37                                                                       {
38     // Unique number to identify the AP
39     ID = ID_in;
40     // X coordinate
41     coordinates[0] = coordinates_in[0];
42     // Y coordinate
43     coordinates[1] = coordinates_in[1];
44     // Height of the AP from the ground
45     height = params.height_AP;
46     // Minimum Contention window size of DCF
47     CW_min = params.CW_min;
```

```
    // Maximum Contention window size of DCF
    CW_max = params.CW_max;
    // slot duration of DCF
    wifi_slot = params.wifi_slot;
    // DIFS duration
    DIFS_time = params.DIFS_time;
    // Maximum transmit opportunity in micro seconds
    MAX_TXOP = params.MAX_TXOP;
    // Downlink load per user in packets per micro second
    load_peruser = params.load_peruser_dl;
    // Total number of STAs associated to this AP
    Connected_UEs = params.n_wifi;
    // Noise floor of the receiver in Watts measured over 1 subcarrier
    Noise = params.Noise;
    // Transmit power of the AP in Watts measured over 1 subcarrier
    Pt_W = params.Pt_AP;
    // Carrier frequency used for Pathloss measurements
    fc = params.fc;
    // Channel Model
    channel_type = params.channel_model;
    // Packet size at the application
    file_size = params.file_size;
    // Wifi Energy Detect Sensing threshold
    WIFI_TH = params.WIFI_TH;
    // Maximum MCS value
    MAX_MCS = params.MAX_MCS;
    // Set the MCS (MAX) for DL transmission for all the STAs
    MCS_DL.insert(MCS_DL.begin(), Connected_UEs, MAX_MCS - 1);
    // bits transmitted in 20micro seconds for each MCS
    WIFI_data_20us.insert(WIFI_data_20us.begin(), &params.WIFI_data_20us[0],
            &params.WIFI_data_20us[MAX_MCS]);
    /*! Data allocated in downlink for each STA
     * This is necessary as there is no explicit signalling of control information
     */
    ALLOC_data_DL.insert(ALLOC_data_DL.begin(), Connected_UEs, 0);
    /*! Data allocated in uplink from each STA
     * This is necessary as there is no explicit signalling of control information
     */
    ALLOC_data_UL.insert(ALLOC_data_UL.begin(), Connected_UEs, 0);
    // Exponential random variable initializer
    std::exponential_distribution<double> exp_distribution(load_peruser);
    // Setting up the buffers and initializing the time first packet arrives
    for (int i = 0; i < Connected_UEs; i++) {
        // create a new linked list for Data buffering for each STA
        std::list<long long> *temp0 = new std::list<long long>;
        // push the list into the Buffer vector
        Buffer_DL.push_back(*temp0);
        // create a new linked list for time file arrived for each STA
        std::list<long long> *temp1 = new std::list<long long>;
        // push the list into the Time file arrived vector
        Time_file_arrived_DL.push_back(*temp1);
        // Find the first packet arrival for a user in DL
         int time_temp = (long long) std::round(exp_distribution(generator));
         // push the time first packet arrives to the vector
         Time_next_file_arrives.push_back(time_temp);
     }
     // Initialize attemp
     attempt.insert(attempt.begin(), Connected_UEs, 0);
     // Initialize success
     success.insert(success.begin(), Connected_UEs, 0);
     // initializ the sum of data for all STA to zero.
     SUM_BUFF_DL.insert(SUM_BUFF_DL.begin(), Connected_UEs, 0);
     // Set the curtrent contention window to the minimum
     CW = CW_min;
}
```

3.1.3.4 void AP::callback_fintx (void ∗ *obj,* std::string *type*)

Callback function for finished tx When a device finishes transmission and notifys the listeners If this device is a part of the listeners then this call back function will be called if the event is FIN_TRANSMITTING This function includes

1. Checking if the calling object is connected to the AP
2. IF so then if AP is already in reception then stops receiving from the STA if it was already receiving from the same STA
3. If the calling object is not associated with AP the it is removed from the interferers list
4. If the AP is already in reception mode then the interference is calculated until this interferer left

INPUT: calling object (obj), type of the calling object (obj.TYPE)

Definition at line 454 of file AP.cpp.

```
454                                             {
455 
456     bool temp = 0;
457     if (type == "AP")
458         temp = 0;
459     if (type == "STA") {
460         // check if the STA is associated with AP
461         for (auto it = BS_UEs.begin(); it != BS_UEs.end(); ++it) {
462             if (obj == *it) {
463                 temp = 1;
464                 break;
465             }
466         }
467     }
468 
469     if (temp && RX_ind == 1 && CURRENT_STA_UL == ((STA*) obj)->ID) { // if STA is associated with AP and AP is receiving
470     // stop receiving
471         RX_ind = 0;
472         // Measure the PL
473         double PL_meas = 0;
474         if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
475             // initialize a vector pointer to measure PL
476             std::vector<double> *PL = new std::vector<double>;
477             // Measure PL from all the interferer sources
478             PL_measure(ACTIVE_Devices, PL, fc, channel_type);
479             for (auto it = PL->begin(); it != PL->end(); ++it) {
480                 // Sum all the received power from all the sources where rx power is tx power*PL
481                 PL_meas += Pt_W * std::pow(10, *it / 10);
482             }
483         } // end of active device
484           // compute interference + noise
485         Int_power += PL_meas * (system_time - local_time)
486                 + Channel_occupy_time * Noise;
487         std::vector<double> d;
488         // calculate the distance between STA and serving AP
489         d.push_back(
490                 std::sqrt(
491                         std::pow(coordinates[0] - ((STA*) obj)->coordinates[0],
492                                 2)
493                                 + std::pow(
494                                         coordinates[1]
495                                                 - ((STA*) obj)->coordinates[1],
496                                         2)
497                                 + std::pow(height - ((STA*) obj)->height, 2)));
498         std::vector<double> *PL_sig = new std::vector<double>;
499         PL_measure(d, PL_sig, fc, channel_type);
500         // calculate the received signal power
501         Sig_power = Channel_occupy_time * Pt_W
502                 * std::pow(10, PL_sig->front() / 10);
503         // Calculate SINR and convert to dB
504         double SINR_db = 10 * std::log10(Sig_power / Int_power);
505         // Log the SINR for stats
506         ((STA*) obj)->SINR.push_back(SINR_db);
507         // find if the transmission is in collsision based on the SINR
508         bool collision = find_prb_collision(SINR_db, ((STA*) obj)->MCS_UL);
509         // update global variable total_tti marking finish of 1 tti
510         total_tti++;
511         if (collision == 0) {
512             // if packet not in collision
513             // update the success
514             ((STA*) obj)->success = ((STA*) obj)->success + 1;
515             // Decrement the data received from the UL buffer
516             ((STA*) obj)->Buffer_UL.front() -= ALLOC_data_UL[((STA*) obj)->ID];
517             // increment the served data STAT
518             ((STA*) obj)->served_data += ALLOC_data_UL[((STA*) obj)->ID];
519             // decrement the total buff
520             ((STA*) obj)->TOT_BUFF -= ALLOC_data_UL[((STA*) obj)->ID];
521             // if the file is finished
522             if (((STA*) obj)->Buffer_UL.front() <= 0) {
523                 // record the time taken for the file to download
524                 int te2 = system_time
525                         - ((STA*) obj)->Time_file_arrived_UL.front();
526                 // push the time taken for STATs
527                 ((STA*) obj)->TTTDeliverOFile_UL.push_back(te2);
528                 // pop the time the file arrived
529                 ((STA*) obj)->Time_file_arrived_UL.pop_front();
530                 // see if there is another file and check if that is scheduled too
531                 if (((STA*) obj)->TOT_BUFF > 0) {
532                     // check the top of the buffer
```

```
                int temp = ((STA*) obj)->Buffer_UL.front();
                // pop it out of the buffer
                ((STA*) obj)->Buffer_UL.pop_front();
                // ADD it to the next packet
                ((STA*) obj)->Buffer_UL.front() += temp;
            } else {
                // due 20us granularity some zero padding will happen
                // if there is no new packet remove the zeros from counting into served data
                ((STA*) obj)->served_data += ((STA*) obj)->TOT_BUFF;
                // remove the zero padding
                ((STA*) obj)->TOT_BUFF = 0;
                // remove the finished packet from buffer
                ((STA*) obj)->Buffer_UL.pop_front();
            } // end of else
              // decrement that file from the BUF_Ind
            ((STA*) obj)->BUF_Ind = ((STA*) obj)->BUF_Ind - 1;
        } // end of buffer_ul
          // MCS Adaptation
          // if total attempts for this STA are 100
        if (((STA*) obj)->attempt == 100) {
            // check among the 100 attempts >70 are success
            if (((STA*) obj)->success > 70) {
                // then increase the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
                // if the MCS is >= MAX MCS
                if (((STA*) obj)->MCS_UL >= MAX_MCS)
                    // then set it to MAX MCS
                    ((STA*) obj)->MCS_UL = MAX_MCS - 1;
            } // end of success
              // check if the success are less than 30%
            else if (((STA*) obj)->success < 30) {
                // then decrement the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
                // if the MCS is less than the minimum
                if (((STA*) obj)->MCS_UL < 0)
                    // then set it to minimum
                    ((STA*) obj)->MCS_UL = 0;
            } // end of if else
              // set the attempts to 0 for this STA
            ((STA*) obj)->attempt = 0;
            // set the success to 0 for this STA
            ((STA*) obj)->success = 0;
        } // end of attempt condition
          // update contention window to the minimum
        ((STA*) obj)->CW = CW_min;
        // reset ALLOC UL
        ALLOC_data_UL[((STA*) obj)->ID] = 0;
    } // end of collision condition
      // if it is collision
    else {
        // double the contention window
        ((STA*) obj)->CW = ((STA*) obj)->CW * 2;
        // if the current window is maximum or greater
        if (((STA*) obj)->CW > CW_max)
            // set it to maximum
            ((STA*) obj)->CW = CW_max;
        // MCS Adaptation
        // if total attempts for this STA are 100
        if (((STA*) obj)->attempt == 100) {
            // check among the 100 attempts >70 are success
            if (((STA*) obj)->success > 70) {
                // then increase the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
                // if the MCS is >= MAX MCS
                if (((STA*) obj)->MCS_UL >= MAX_MCS)
                    // then set it to MAX MCS
                    ((STA*) obj)->MCS_UL = MAX_MCS - 1;
            } // end of success
              // check if the success are less than 30%
            else if (((STA*) obj)->success < 30) {
                // then decrement the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
                // if the MCS is less than the minimum
                if (((STA*) obj)->MCS_UL < 0)
                    // then set it to minimum
                    ((STA*) obj)->MCS_UL = 0;
            } // end of if else
              // set the attempts to 0 for this STA
            ((STA*) obj)->attempt = 0;
            // set the success to 0 for this STA
            ((STA*) obj)->success = 0;
        } // end of attempt condition
    }
    // reset ALLOC UL
    ALLOC_data_UL[((STA*) obj)->ID] = 0;
} // end of if temp
  // if EVENT from associated STA and not receiving this STA
```

```
    else if (temp) {
        // double the contention window
        ((STA*) obj)->CW = ((STA*) obj)->CW * 2;
        // if the current window is maximum or greater
        if (((STA*) obj)->CW > CW_max)
            // set it to maximum
            ((STA*) obj)->CW = CW_max;
        // MCS Adaptation
        // if total attempts for this STA are 100
        if (((STA*) obj)->attempt == 100) {
            // check among the 100 attempts >70 are success
            if (((STA*) obj)->success > 70) {
                // then increase the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
                // if the MCS is >= MAX MCS
                if (((STA*) obj)->MCS_UL >= MAX_MCS)
                    // then set it to MAX MCS
                    ((STA*) obj)->MCS_UL = MAX_MCS - 1;
            } // end of success
              // check if the success are less than 30%
            else if (((STA*) obj)->success < 30) {
                // then decrement the MCS
                ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
                // if the MCS is less than the minimum
                if (((STA*) obj)->MCS_UL < 0)
                    // then set it to minimum
                    ((STA*) obj)->MCS_UL = 0;
            } // end of if else
              // set the attempts to 0 for this STA
            ((STA*) obj)->attempt = 0;
            // set the success to 0 for this STA
            ((STA*) obj)->success = 0;
        } // end of attempt condition
          // reset ALLOC UL
        ALLOC_data_UL[((STA*) obj)->ID] = 0;
        // find the distance from the AP
        double d = std::sqrt(
                std::pow(coordinates[0] - ((STA*) obj)->coordinates[0], 2)
                        + std::pow(
                                coordinates[1] - ((STA*) obj)->coordinates[1],
                                2)
                        + std::pow(height - ((STA*) obj)->height, 2));
        if (RX_ind == 1) {
            // Measure the PL
            double PL_meas = 0;
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type);
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    PL_meas += Pt_W * std::pow(10, *it / 10);
                }
            } // end of active device
              // Add the interferece power of previous state
            Int_power += PL_meas * (system_time - local_time);
            // record the time the interference changed
            local_time = system_time;
            // remove the STA from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
                    ACTIVE_Devices.erase(it);
                    break;
                }
            } // end of for
        } // end of if
        else {
            // remove the STA from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
                    ACTIVE_Devices.erase(it);
                    break;
                }
            } // end of for
        } // end of else
    } // end of elseif
    else {
        double d = 0;
        if (type == "STA")
            d = std::sqrt(
```

```
                    std::pow(coordinates[0] - ((STA*) obj)->coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((STA*) obj)->coordinates[1], 2)
                            + std::pow(height - ((STA*) obj)->height, 2));
        if (type == "AP")
            d = std::sqrt(
                    std::pow(coordinates[0] - ((AP*) obj)->coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((AP*) obj)->coordinates[1], 2)
                            + std::pow(height - ((AP*) obj)->height, 2));

        if (RX_ind == 1) {
            // Measure the PL
            double PL_meas = 0;
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type);
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    PL_meas += Pt_W * std::pow(10, *it / 10);
                }
            } // end of active device
              // Add the interferece power of previous state
            Int_power += PL_meas * (system_time - local_time);
            // record the time the interference changed
            local_time = system_time;
            // remove the device from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
                    ACTIVE_Devices.erase(it);
                    break;
                }
            } // end of for
        } else {
            // remove the device from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
                    ACTIVE_Devices.erase(it);
                    break;
                }
            }
        }
    }
}
```

3.1.3.5 void AP::callback_tx (void * *obj*, std::string *type*)

call backs

Callback function for tx When a device transmits and notifys the listeners If this device is a part of the listeners then this call back function will be called if the event is TRANSMITTING This function includes

1. Checking if the calling object is connected to the AP
2. IF so then if AP is not already in reception then starts receiving from the STA
3. If the calling object is not associated with AP the it is put in the interferers list
4. If the AP is already in reception mode then the interference is calculated until the new interferer came

INPUT: calling object (obj), type of the calling object (obj.TYPE)

Definition at line 361 of file AP.cpp.

```
361                                               {
362     // temperoray variable
363     bool temp = 0;
```

```

    if (type == "AP")
        // is EVENT from AP
        temp = 0;
    if (type == "STA") {
        // check if the STA is associated with AP
        for (auto it = BS_UEs.begin(); it != BS_UEs.end(); ++it) {
            if (obj == *it) {
                temp = 1;
                break;
            }
        } // end of for
    } // end if
    if (temp && TX_ind != 1 && RX_ind != 1) {
        // if the EVENT is from connected STA and AP is not receiving not transmitting
        // Start receiving
        RX_ind = 1;
        // stop contending
        contending = 0;
        // record the STA ID
        CURRENT_STA_UL = ((STA*) obj)->ID;
        // Mark the time the transmission started
        local_time = system_time;
        // init sig power
        Sig_power = 0;
        // init interferer power
        Int_power = 0;
        // Record the channel occupy time
        Channel_occupy_time = ((STA*) obj)->Channel_occupy_time;
    } // end of temp
    else { // else add the event sources to interferers list
           // if the event is from other STA
        double d = 0;
        if (type == "STA")
            // find the distance from the interferer
            d = std::sqrt(
                    std::pow(coordinates[0] - ((STA*) obj)->
  coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((STA*) obj)->coordinates[1], 2)
                            + std::pow(height - ((STA*) obj)->height, 2));
        if (type == "AP")
            // measure distance from AP
            d = std::sqrt(
                    std::pow(coordinates[0] - ((AP*) obj)->
  coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((AP*) obj)->coordinates[1], 2)
                            + std::pow(height - ((AP*) obj)->height, 2));

        // check if the current AP is in receiving mode
        if (RX_ind == 1) {
            // Measure the PL
            double PL_meas = 0;
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type);
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    PL_meas += Pt_W * std::pow(10, *it / 10);
                }
            } // end of active devices
              // Add the interferece power of previous state
            Int_power += PL_meas * (system_time - local_time);
            // record the time the interference changed
            local_time = system_time;
            // push the interferer distance to interferers list
            ACTIVE_Devices.push_back(d);
        } // end of RX condition
        else {
            // Add the new interferer to the interferers list
            ACTIVE_Devices.push_back(d);
        } // end of else
    } // end of else
} // end of function
```

3.1.3.6 int AP::check_buffer ( )

check the buffers for packet arrival If there is a packet arrived then starts contending Does energy measurement for 1 micro second

Definition at line 137 of file AP.cpp.

```
137                  {
138     int jump = 0, i = 0;
139     // exponential distribution for load
140     std::exponential_distribution<double> exp_distribution(load_peruser);
141     // Decrement time next file arrives
142     for (auto it = Time_next_file_arrives.begin();
143             it != Time_next_file_arrives.end(); ++it) {
144         (*it) -= 1;
145         if (*it <= 0) { // Check if the file arrived
146             // push the time the file arrived
147             Time_file_arrived_DL[i].push_back(system_time);
148             // Find the next packet arrival time
149             Time_next_file_arrives[i] = (long long) std::round(
150                     exp_distribution(generator));
151             // Push the arrived file into the buffer
152             Buffer_DL[i].push_back(file_size);
153             // Sum the file size to sum buff buffer size
154             SUM_BUFF_DL[i] += file_size;
155             // Increment the STATISTIC total files arrived
156             total_files += 1;
157             // Increment buffer indicator
158             BUF_Ind += 1;
159             // Sum the file size to total buffer size
160             TOT_BUFF += file_size;
161         }
162         i++; // Next user
163     }
164     // update jump with the time next file arrives
165     jump = Time_next_file_arrives[0];
166     // find the minimum among all the users buffer
167     for (auto it = Time_next_file_arrives.begin();
168             it != Time_next_file_arrives.end(); ++it) {
169         jump = jump < *it ? jump : *it;
170     }
171 
172     if (contending == 0 && TX_ind == 0 && RX_ind == 0) {
173         // if not TX and not Contending and not RX
174         // then check the buffer and if there is data start contending
175         if (BUF_Ind > 0) {
176             // As there is data start contending
177             contending = true;
178             // set jump to zero as there is data to tx
179             jump = 0;
180             // set a uniform distribution to get values between [0 CW-1]
181             std::uniform_int_distribution<int> uni_distribution(0, CW - 1);
182             // get a random number from uniform distribution and multiply it with wifi_slot to get time in
       us
183             Counter = uni_distribution(generator) * wifi_slot;
184             // Set DIFS_Ind to zero as this is the start of DCF procedure
185             DIFS_Ind = 0;
186             // Set the DIFS Counter to DIFS duration
187             DIFS_counter = DIFS_time;
188             // Initialize energy measurement to thermal noise
189             Energy_meas = Noise;
190             // Measure the energy from all interferers
191             if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
192                 // initialize a vector pointer to measure PL
193                 std::vector<double> *PL = new std::vector<double>;
194                 // Measure PL from all the interferer sources
195                 PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
196                 for (auto it = PL->begin(); it != PL->end(); ++it) {
197                     // Sum all the received power from all the sources where rx power is tx power*PL
198                     Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
199                 } // end of for
200             } // end of if
201         } // end of if
202     } // end of if
203     else if (contending == 1) {
204         // if contending then do energy measurements
205         // set jump to 0
206         jump = 0;
207         // Initialize energy measurement to thermal noise
208         Energy_meas = Noise;
209         // Measure the energy from all interferers
210         if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
211             // initialize a vector pointer to measure PL
212             std::vector<double> *PL = new std::vector<double>;
213             // Measure PL from all the interferer sources
```

```
214             PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
215             for (auto it = PL->begin(); it != PL->end(); ++it) {
216                 // Sum all the received power from all the sources where rx power is tx power*PL
217                 Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
218             } // end of for
219         } // end of if
220     } // end of if else
221     else {
222         // might be receiving might be transmitting
223         // set the jump to zero
224         jump = 0;
225     } // end of else
226       // return time jump value
227     return jump;
228 }
```

3.1.3.7 void AP::check_tx_time ( )

check tx duration and decrement channel occupancy time when transmission is over notify all the listening nodes

Definition at line 334 of file AP.cpp.

```
334                        {
335     // check if STA is tranmitting
336     if (TX_ind == 1) {
337         // Decrement the channel occupy time
338         Channel_occupy_time = Channel_occupy_time - 1;
339         // check if channel occupy time is zero
340         if (Channel_occupy_time <= 0) {
341             // set tx ind to 0
342             TX_ind = 0;
343             // notify the listener group about finished transmission
344             notify_listener(FIN_TRANSMITTING);
345         } // end of chanel occupy cond
346     } // end of tx ind condition
347 } // end of function
```

3.1.3.8 void AP::jump_time (int *jump*)

used to do a time jump is all the nodes dont have data to tx

jump time if no data available at all the nodes

Definition at line 125 of file AP.cpp.

```
125                            {
126     // Substract jump time from the time next file arrives
127     for (auto it = Time_next_file_arrives.begin();
128             it != Time_next_file_arrives.end(); ++it) {
129         (*it) -= jump;
130     }
131 }
```

3.1.3.9 void AP::notify_listener (int *EVENT*)

Notify the listeners This function calls the callback functions of the listeners based on the event registered INPUT: EVENT

Definition at line 773 of file AP.cpp.

```
773                                   {
774     switch (EVENT) {
775 
776     case TRANSMITTING:
777         // Call back AP
778         for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
779             ((AP*) (*it))->callback_tx(this, this->TYPE);
780         }
781 
782         // call back STA
```

```
        for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
            ((STA*) (*it))->callback_tx(this, this->TYPE);
        }

        break;

    case FIN_TRANSMITTING:
        // AP call back
        for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
            ((AP*) (*it))->callback_fintx(this, this->TYPE);
        }

        // STA call back
        for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
            ((STA*) (*it))->callback_fintx(this, this->TYPE);
        }

        break;
    default:
        std::cout << "unknown event occured \n";
    } // end of switch

} // end of function
```

3.1.3.10 void AP::sched_tx ( )

sched and transmit Based on the energy measuremnt from previous function Clear channel assessment is done based on CCA DCF procedures are done. If AP gets the channel then ot schedules a UE in round robin fashion and transmits

Definition at line 236 of file AP.cpp.

```
236                  {
237     // DO DCF if contending
238     if (contending == 1) {
239         // See if DIFS is done
240         if (DIFS_Ind == 1) {
241             // check if the measured enrgy is less than WIFI sensing threshold
242             if (Energy_meas > WIFI_TH) {
243                 // If energy meas id > sens thresh then bring the counter to multiple of wifi slots
244                 Counter = Counter - (Counter % wifi_slot) + wifi_slot;
245                 // Set DIFS Ind to 0
246                 DIFS_Ind = 0;
247                 // Set the DIFS counter
248                 DIFS_counter = DIFS_time;
249             } // end of if energy condition
250             else
251                 // Decrement the counter by 1 us If energy meas is less than sensing threshold
252                 Counter = Counter - 1;
253             // Check whether the counter is zero
254             if (Counter <= 0) {
255                 // If counter is zero
256                 // set contending to false
257                 contending = 0;
258                 // Set DIFS indicator to zero
259                 DIFS_Ind = 0;
260                 // Set DIFS counter
261                 DIFS_counter = DIFS_time;
262                 // Set TX indicater to 1
263                 TX_ind = 1;
264                 // Start Scheduling
265                 // Set channel occupy time to zero;
266                 Channel_occupy_time = 0;
267                 // Do round Robin
268                 if (CURRENT_STA_DL == Connected_UEs - 1) { // if current UE is last one pick
     the next one
269                     CURRENT_STA_DL = 0; // pick the first UE
270                 } else {
271                     CURRENT_STA_DL += 1; // pick the next UE
272                 } // end of else
273                   // schedule the UE
274                 while (Channel_occupy_time == 0) {
275                     // check if the current UE has data
276                     if (SUM_BUFF_DL[CURRENT_STA_DL] > 0) {
277                         // increment the attempt
278                         attempt[CURRENT_STA_DL] = attempt[
      CURRENT_STA_DL] + 1;
279                         // check the time required
280                         long long time =
281                                 std::ceil(
282                                         (double) SUM_BUFF_DL[
```

```
CURRENT_STA_DL]
                                                / WIFI_data_20us[MCS_DL[
CURRENT_STA_DL]]);
                            // check whether the time required is less than the MAX tx opportunity
                            if (20 * time < MAX_TXOP) { // 20us is the minimum granularity
                                // update the channel occupy time
                                Channel_occupy_time = 20 * time;
                                // Update Allocated data
                                ALLOC_data_DL[CURRENT_STA_DL] =
                                        SUM_BUFF_DL[CURRENT_STA_DL];
                                // update the pumped data
                                pumped_data += SUM_BUFF_DL[
CURRENT_STA_DL];
                            } else {
                                // occupy the channel for max tx op
                                Channel_occupy_time = MAX_TXOP;
                                // update Allocated data
                                ALLOC_data_DL[CURRENT_STA_DL] =
                                        WIFI_data_20us[MCS_DL[CURRENT_STA_DL]]
                                                * MAX_TXOP / 20;
                                // update pumped data
                                pumped_data +=
                                        WIFI_data_20us[MCS_DL[CURRENT_STA_DL]]
                                                * MAX_TXOP / 20;
                            } // end of else
                        } else {
                            // Do round Robin
                            // if current UE is last one pick the next one
                            if (CURRENT_STA_DL == Connected_UEs - 1)
                                CURRENT_STA_DL = 0; // pick the first UE
                            else
                                CURRENT_STA_DL += 1; // pick the next UE
                        } // end of else
                    } // end of inf while
                      // Notify the listening group about this transmission
                    notify_listener(TRANSMITTING);
                } // end of counter zero
            } else if (Energy_meas < WIFI_TH) { // if measured energy is less than the WIFI sensing threshold
 (1us)
                // Decrement DIFS counter by 1us
                DIFS_counter = DIFS_counter - 1;
                // Check if the DIFS counter is zero
                if (DIFS_counter <= 0) {
                    // Indicate that DIFS is done
                    DIFS_Ind = 1;
                    // initialize the DIFS counter (Redundant but to be sure)
                    DIFS_counter = DIFS_time;
                } // end of if
            } // end of enregy meas
        } // End of if contending
} // End of function
```

3.1.4 Member Data Documentation

3.1.4.1 std::vector<long long> AP::ALLOC_data_DL

Data allocated in downlink for each STA This is necessary as there is no explicit signalling of control information

Definition at line 116 of file AP.h.

3.1.4.2 std::vector<long long> AP::ALLOC_data_UL

Data allocated in uplink from each STA This is necessary as there is no explicit signalling of control information

Definition at line 120 of file AP.h.

3.1.4.3 long long AP::BUF_Ind = 0

Total files yet to be transmitted 0 if no data !! >0 if data counts currnt files

Definition at line 135 of file AP.h.

3.1.4.4 int AP::ID

Device ID and Device TYPE combined will give a unique ID to the device So there is no need for a physical MAC Address Device ID

Definition at line 94 of file AP.h.

3.1.4.5 std::vector<void *> AP::listener_STA

Event listeners vector containing the station object handles

Definition at line 160 of file AP.h.

3.1.4.6 bool AP::RX_ind = 0

Reception status indicator 0 if not rx 1 if rx

Definition at line 126 of file AP.h.

3.1.4.7 bool AP::TX_ind = 0

Transmission status indicator 0 if not tx 1 if tx

Definition at line 123 of file AP.h.

The documentation for this class was generated from the following files:

- /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/AP.h
- /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/AP.cpp

3.2 parameters Struct Reference

Public Attributes

- int NCell
- int n_wifi

  Total number of nodes per cell.
- double BW

  System BandWidth.
- double dimensions [2]

  Dimension of a single cell.
- double height_STA

  Height of the station.
- double height_AP

  height of the AP
- std::string channel_model

  Channel Model.
- int CW_min
- int CW_max

  Maximum Contention Window.
- int wifi_slot

  Wifi Slot duration.
- int DIFS_time

  DIFS Time.

- long long MAX_TXOP

  Maximum transmit opportunity.
- double DL_UL
- double DL_percent

  DL perecentage.
- double UL_percent

  UL perecentage.
- double Network_load_bps

  Network load in bps.
- double load_peruser_dl

  load per user in dl in files per micro second
- double load_peruser_ul

  load per user in ul in files per micro second
- double Noise

  Receiver noise floor.
- double Pt_AP

  AP Transmit Power.
- double Pt_STA

  STA Transmit Power.
- int fc

  Operating frequency.
- long long file_size

  file size in bits
- double WIFI_TH

  Sensing threshold in Watts measured over 1 subcarrier.
- std::vector< double > WIFI_data_20us

  bits transmitted in 20micro seconds for each MCS
- int MAX_MCS

  Total MCS available.

3.2.1 Detailed Description

Definition at line 44 of file common.h.

3.2.2 Member Data Documentation

3.2.2.1 int parameters::CW_min

DCF parameters Minimum contention window

Definition at line 62 of file common.h.

3.2.2.2 double parameters::DL_UL

load stats DL to UL Ratio

Definition at line 73 of file common.h.

3.2.2.3 int parameters::NCell

System Specific Total cells in the layout

Definition at line 47 of file common.h.

The documentation for this struct was generated from the following file:

- /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/common.h

3.3 STA Class Reference

```
#include <STA.h>
```

Public Member Functions

- STA ()

  METHODS.
- void STA_init (int ID, double coordinates[2], parameters params, void ∗BS_obj)
- void jump_time (int jump)
- int check_buffer ()
- void sched_tx ()
- void check_tx_time ()
- void callback_tx (void ∗obj, std::string type)

  call backs
- void callback_fintx (void ∗obj, std::string type)
- void add_listener (void ∗obj, std::string type)

  framework support functions
- void notify_listener (int EVENT)

Public Attributes

- int ID
- std::string TYPE = "STA"

  Device Type.
- double height

  AP height from the ground.
- double coordinates [2]

  AP coordintes [x,y].
- int MCS_UL

  UL MCS.
- int MAX_MCS

  Total MCS Available.
- std::vector< double > ACTIVE_Devices

  Vector containing the distance of all the Active devices int the network.
- std::vector< double > ACTIVE_Devices_power

  Vector containing the tx power of all the Active devices int the network.
- bool TX_ind = 0
- bool RX_ind = 0
- double Pt_W

  Transmit power of the AP in Watts measured over 1 subcarrier.
- std::list< long long > Buffer_UL

list of files : mimicks the packet que

- int BUF_Ind = 0
- long long TOT_BUFF = 0

  Total data yet to be transmitted.
- int Channel_occupy_time = 0

  channel occupy time for transmission
- std::list< long long > Time_file_arrived_UL

  list of time a file arrived per user: complements Buffer U0L
- std::vector< long long > TTTDeliverOFile_UL

  vector containing Time in micro seconds took to delive a file in IL used for STAISTICS
- std::vector< double > SINR

  vector containing SINR in dB in UL for each transmission used for STAISTICS
- int total_files = 0

  total files received used for STATISTICS
- long long served_data = 0

  Total served data in UL used for STATISTICS.
- long long pumped_data = 0

  Total data pumped into the channel in UL used for STATISTICS.
- long long attempt = 0

  Total attempts : Used for MCS adaptation.
- long long success = 0

  Total success : Used for MCS adaptation.
- int CW

  current contention window size
- std::vector< void * > listener_STA
- std::vector< void * > listener_AP

  vector containing the access point object handles

3.3.1 Detailed Description

Class STA IEEE 802.11ac Implements 802.11 DCF PHY is completely Abstracted Station node contains a traffic generator + lower MAC + PHY abstraction

Definition at line 28 of file STA.h.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 STA::STA ( )

METHODS.

Constructor

Definition at line 29 of file STA.cpp.

```
29        {
30 }
```

3.3.3 Member Function Documentation

3.3.3.1 void STA::add_listener (void ∗ *obj*, std::string *type*)

framework support functions

ADD listeners This function adds a STA/AP to the listeners group INPUT: object handle (obj) , object type (obj.TYPE)

ADD listener type station This function adds a STA/AP to the listeners group INPUT: object handle (obj) , object type (obj.TYPE)

Definition at line 670 of file STA.cpp.

```
670                                               {
671     // if the listener is AP
672     if (type == "AP")
673         // ADD to AP group
674         listener_AP.push_back(obj);
675     // if the listener is STA
676     if (type == "STA")
677         // ADD to STA group
678         listener_STA.push_back(obj);
679 }
```

3.3.3.2 void STA::callback_fintx (void ∗ *obj*, std::string *type*)

Callback function for finished tx When a device finishes transmission and notifys the listeners If this device is a part of the listeners then this call back function will be called if the event is FIN_TRANSMITTING This function includes

1. Checking if the calling object is the connected AP
2. IF so then if STA is already in reception then stops receiving from the AP
3. If the calling object is not the associated AP then it is removed from the interferers list
4. If the STA is already in reception mode then the interference is calculated until this new interferer left

INPUT: calling object (obj), type of the calling object (obj.TYPE)

Callback function for finished tx When a device finishes transmission and notifys the listeners If this device is a part of the listeners then this call back function will be called if the event is FIN_TRANSMITTING This function includes

1. Checking if the calling object is connected to the AP
2. IF so then if AP is already in reception then stops receiving from the STA if it was already receiving from the same STA
3. If the calling object is not associated with AP the it is removed from the interferers list
4. If the AP is already in reception mode then the interference is calculated until this interferer left

INPUT: calling object (obj), type of the calling object (obj.TYPE)

Definition at line 393 of file STA.cpp.

```
393                                                 {
394     bool temp = 0;
395     if (type == "STA")
396         temp = 0;
397     if (type == "AP") {
398         temp = (((AP*) obj)->ID == ((AP*) BS_obj)->ID);
399     }
400     // if EVENT from serving AP and STA is in receiving mode and data is intended for this STA
401     if (temp && RX_ind == 1 && ((AP*) obj)->CURRENT_STA_DL == ID) {
402         // stop receiving
403         RX_ind = 0;
404         // Measure the PL
```

```
double PL_meas = 0;
if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
    // initialize a vector pointer to measure PL
    std::vector<double> *PL = new std::vector<double>;
    // Measure PL from all the interferer sources
    PL_measure(ACTIVE_Devices, PL, fc, channel_type);
    for (auto it = PL->begin(); it != PL->end(); ++it) {
        // Sum all the received power from all the sources where rx power is tx power*PL
        PL_meas += Pt_W * std::pow(10, *it / 10);
    }
} // end of active device
  // compute interference + noise
Int_power += PL_meas * (system_time - local_time)
        + Channel_occupy_time * Noise;
std::vector<double> d;
// calculate the distance between STA and serving AP
d.push_back(
        std::sqrt(
                std::pow(coordinates[0] - ((AP*) obj)->
coordinates[0],
                        2)
                        + std::pow(
                                coordinates[1]
                                        - ((AP*) obj)->coordinates[1],
                                2)
                        + std::pow(height - ((AP*) obj)->height, 2)));
std::vector<double> *PL_sig = new std::vector<double>;
PL_measure(d, PL_sig, fc, channel_type);
// calculate the received signal power
Sig_power = Channel_occupy_time * Pt_W
        * std::pow(10, PL_sig->front() / 10);
// Calculate SINR and convert to dB
double SINR_db = 10 * std::log10(Sig_power / Int_power);
// Log the SINR for stats
((AP*) obj)->SINR.push_back(SINR_db);
// find if the transmission is in collsision based on the SINR
bool collision = find_prb_collision(SINR_db, ((AP*) obj)->MCS_DL[ID]);
// update global variable total_tti marking finish of 1 tti
total_tti++;
if (collision == 0) {
    // if packet not in collision
    // update the success
    ((AP*) obj)->success[ID] += 1;
    // Decrement the data received from the DL buffer
    ((AP*) obj)->Buffer_DL[ID].front() -=
            ((AP*) obj)->ALLOC_data_DL[ID];
    // Decrement the data received from the summ BUFF DL buffer
    ((AP*) obj)->SUM_BUFF_DL[ID] -= ((AP*) obj)->ALLOC_data_DL[ID];
    // increment the served data STAT
    ((AP*) obj)->served_data += ((AP*) obj)->ALLOC_data_DL[ID];
    // decrement the total buff
    ((AP*) obj)->TOT_BUFF -= ((AP*) obj)->ALLOC_data_DL[ID];
    // if the file is finished
    if (((AP*) obj)->Buffer_DL[ID].front() <= 0) {
        // record the time taken for the file to download
        int te2 = system_time
                - ((AP*) obj)->Time_file_arrived_DL[ID].front();
        // push the time taken for STATs
        ((AP*) obj)->TTTDeliverOFile_DL.push_back(te2);
        // pop the time the file arrived
        ((AP*) obj)->Time_file_arrived_DL[ID].pop_front();
        // see if there is another file and check if that is scheduled too
        if (((AP*) obj)->SUM_BUFF_DL[ID] > 0) {
            // check the top of the buffer
            int temp = ((AP*) obj)->Buffer_DL[ID].front();
            // pop it out of the buffer
            ((AP*) obj)->Buffer_DL[ID].pop_front();
            // ADD it to the next packet
            ((AP*) obj)->Buffer_DL[ID].front() += temp;
        } else {
            // due 20us granularity some zero padding will happen
            // if there is no new packet remove the zeros from counting into served data
            ((AP*) obj)->served_data += ((AP*) obj)->SUM_BUFF_DL[ID];
            // do the same for tot buff
            ((AP*) obj)->TOT_BUFF -= ((AP*) obj)->SUM_BUFF_DL[ID];
            // set the sum buff to zero
            ((AP*) obj)->SUM_BUFF_DL[ID] = 0;
            // remove the finished packet from buffer
            ((AP*) obj)->Buffer_DL[ID].pop_front();
        } // end of else
          // decrement that file from the BUF_Ind
        ((AP*) obj)->BUF_Ind -= 1;
    } // end of buffer_dl
      // reset the ALLOC data dl
    ((AP*) obj)->ALLOC_data_DL[ID] = 0;
    // MCS Adaptation
    // if total attempts for this STA are 100
```

```
            if (((AP*) obj)->attempt[ID] == 100) {
                // check among the 100 attempts >70 are success
                if (((AP*) obj)->success[ID] > 70) {
                    // then increase the MCS
                    ((AP*) obj)->MCS_DL[ID] += 1;
                    // if the MCS is >= MAX MCS
                    if (((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
                        // then set it to MAX MCS
                        ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
                } // end of success
                  // check if the success are less than 30%
                else if (((AP*) obj)->success[ID] < 30) {
                    // then decrement the MCS
                    ((AP*) obj)->MCS_DL[ID] -= 1;
                    // if the MCS is less than the minimum
                    if (((AP*) obj)->MCS_DL[ID] < 0)
                        // then set it to minimum
                        ((AP*) obj)->MCS_DL[ID] = 0;
                } // end of if else
                  // set the attempts to 0 for this STA
                ((AP*) obj)->attempt[ID] = 0;
                // set the success to 0 for this STA
                ((AP*) obj)->success[ID] = 0;
            } // end of attempt condition
              // reset the ALLCO data dl
            ((AP*) obj)->ALLOC_data_DL[ID] = 0;
            // update contention window to the minimum
            ((AP*) obj)->CW = CW_min;
        } // end of collision condition
          // if it is collision
        else {
            // double the contention window
            ((AP*) obj)->CW = ((AP*) obj)->CW * 2;
            // if the current window is maximum or greater
            if (((AP*) obj)->CW > CW_max)
                // set it to maximum
                ((AP*) obj)->CW = CW_max;
            // MCS Adaptation
            // if total attempts for this STA are 100
            if (((AP*) obj)->attempt[ID] == 100) {
                // check among the 100 attempts >70 are success
                if (((AP*) obj)->success[ID] > 70) {
                    // then increase the MCS
                    ((AP*) obj)->MCS_DL[ID] += 1;
                    // if the MCS is >= MAX MCS
                    if (((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
                        // then set it to MAX MCS
                        ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
                } // end of success
                  // check if the success are less than 30%
                else if (((AP*) obj)->success[ID] < 30) {
                    // then decrement the MCS
                    ((AP*) obj)->MCS_DL[ID] -= 1;
                    // if the MCS is less than the minimum
                    if (((AP*) obj)->MCS_DL[ID] < 0)
                        // then set it to minimum
                        ((AP*) obj)->MCS_DL[ID] = 0;
                } // end of if else
                  // set the attempts to 0 for this STA
                ((AP*) obj)->attempt[ID] = 0;
                // set the success to 0 for this STA
                ((AP*) obj)->success[ID] = 0;
            } // end of attempt condition
        }
        // reset the ALLCO data dl
        ((AP*) obj)->ALLOC_data_DL[ID] = 0;
    }
    // if EVENT from serving AP and STA is not in receiving mode and data is intended for this STA
    else if (temp && ((AP*) obj)->CURRENT_STA_DL == ID) {
        // double the contention window
        ((AP*) obj)->CW = ((AP*) obj)->CW * 2;
        // if the current window is maximum or greater
        if (((AP*) obj)->CW > CW_max)
            // set it to maximum
            ((AP*) obj)->CW = CW_max;
        // MCS Adaptation
        // if total attempts for this STA are 100
        if (((AP*) obj)->attempt[ID] == 100) {
            // check among the 100 attempts >70 are success
            if (((AP*) obj)->success[ID] > 70) {
                // then increase the MCS
                ((AP*) obj)->MCS_DL[ID] += 1;
                // if the MCS is >= MAX MCS
                if (((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
                    // then set it to MAX MCS
                    ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
            } // end of success
```

```
                // check if the success are less than 30%
            else if (((AP*) obj)->success[ID] < 30) {
                // then decrement the MCS
                ((AP*) obj)->MCS_DL[ID] -= 1;
                // if the MCS is less than the minimum
                if (((AP*) obj)->MCS_DL[ID] < 0)
                    // then set it to minimum
                    ((AP*) obj)->MCS_DL[ID] = 0;
            } // end of if else
              // set the attempts to 0 for this STA
            ((AP*) obj)->attempt[ID] = 0;
            // set the success to 0 for this STA
            ((AP*) obj)->success[ID] = 0;
        } // end of attempt condition
          // find the distance from the AP
        double d = std::sqrt(
                std::pow(coordinates[0] - ((AP*) obj)->coordinates[0], 2)
                        + std::pow(coordinates[1] - ((AP*) obj)->coordinates[1],
                                2) + std::pow(height - ((AP*) obj)->height, 2));
        // remove the AP from the interferers list
        for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                ++it) {
            if (*it == d) {
                ACTIVE_Devices.erase(it);
                break;
            } // end of if loop
        } // end of for loop
    } // end of else if
    else {
        double d = 0;
        // if EVENT from STA
        if (type == "STA")
            // Measure the distance
            d = std::sqrt(
                    std::pow(coordinates[0] - ((STA*) obj)->coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((STA*) obj)->coordinates[1], 2)
                            + std::pow(height - ((STA*) obj)->height, 2));
        if (type == "AP") // if EVENT from AP
            // Measure the distance
            d = std::sqrt(
                    std::pow(coordinates[0] - ((AP*) obj)->coordinates[0], 2)
                            + std::pow(
                                    coordinates[1]
                                            - ((AP*) obj)->coordinates[1], 2)
                            + std::pow(height - ((AP*) obj)->height, 2));

        // if STA is in RX mode
        if (RX_ind == 1) {
            // Measure the PL
            double PL_meas = 0;
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type);
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    PL_meas += Pt_W * std::pow(10, *it / 10);
                }
            } // end of active device
              // accumulate interference power
            Int_power += PL_meas * (system_time - local_time);
            // record the interference change time
            local_time = system_time;
            // remove the device from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
                    ACTIVE_Devices.erase(it);
                    break;
                }
            } // end of for loop
        } // end of RX=1
        else {
            // remove the device from the interferers list
            for (auto it = ACTIVE_Devices.begin(); it != ACTIVE_Devices.end();
                    ++it) {
                if (*it == d) {
```

```
                        ACTIVE_Devices.erase(it);
                        break;
                    }
                } // end of for loop
            } // end of else
        } // end of else
    } // end of function
```

3.3.3.3 void STA::callback_tx (void * *obj,* std::string *type*)

call backs

Callback function for tx When a device transmits and notifys the listeners If this device is a part of the listeners then this call back function will be called if the event is TRANSMITTING This function includes

1. Checking if the calling object is the AP
2. IF so then if STA is not already in Transmiting state then starts receiving from the AP
3. If the calling object is not the associated AP then it is put in the interferers list
4. If the STA is already in reception mode then the interference is calculated until this new interferer came

INPUT: calling object (obj), type of the calling object (obj.TYPE)

Definition at line 306 of file STA.cpp.

```
306                                                 {
307     // temperoray variable
308     bool temp = 0;
309     if (type == "STA")
310         // is EVENT from STA
311         temp = 0;
312     if (type == "AP") {
313         // is EVENT from AP
314         // check if the AP is the serving AP
315         temp = (((AP*) obj)->ID == ((AP*) BS_obj)->ID);
316     } // end of type
317     if (temp && TX_ind != 1 && RX_ind != 1
318             && ((AP*) obj)->CURRENT_STA_DL == ID) {
319         // if the EVENT is from serving AP and STA not receiving not transmitting and Transmission is intended for this STA
320         // Start receiving
321         RX_ind = 1;
322         // stop contending
323         contending = 0;
324         // Mark the time the transmission started
325         local_time = system_time;
326         // init sig power
327         Sig_power = 0;
328         // init interferer power
329         Int_power = 0;
330         // Record the channel occupy time
331         Channel_occupy_time = ((AP*) obj)->Channel_occupy_time;
332     } // end of temp
333       // else add the event sources to interferers list
334     else {
335         double d = 0;
336         // if the event is from other STA
337         if (type == "STA")
338             // find the distance from the interferer
339             d = std::sqrt(
340                     std::pow(coordinates[0] - ((STA*) obj)->coordinates[0], 2)
341                             + std::pow(
342                                     coordinates[1]
343                                             - ((STA*) obj)->coordinates[1], 2)
344                             + std::pow(height - ((STA*) obj)->height, 2));
345         if (type == "AP")
346             // find the distance from the interferer
347             d = std::sqrt(
348                     std::pow(coordinates[0] - ((AP*) obj)->coordinates[0], 2)
349                             + std::pow(
350                                     coordinates[1]
351                                             - ((AP*) obj)->coordinates[1], 2)
352                             + std::pow(height - ((AP*) obj)->height, 2));
```

```
        // check if the current STA is in receiving mode
        if (RX_ind == 1) {
            // Measure the PL
            double PL_meas = 0;
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type);
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    PL_meas += Pt_W * std::pow(10, *it / 10);
                }
            } // end of active device
              // Add the interference power of previous state
            Int_power += PL_meas * (system_time - local_time);
            // record the time the interference changed
            local_time = system_time;
            // push the interferer distance to interferers list
            ACTIVE_Devices.push_back(d);
        } // end of RX condition
        else {
            // Add the new interferer to the interferers list
            ACTIVE_Devices.push_back(d);
        } // end of else condition
    } // end of else not associated
} // end of the function
```

3.3.3.4 int STA::check_buffer ( )

check the buffers for packet arrival If there is a packet arrived then starts contending Does energy measurement for 1 micro second

check the buffers for packet arrival If there is a packet arrived then starts contending Does energy measurement for 1 micro second OUTPUT: time jump value (int)

Definition at line 105 of file STA.cpp.

```
105                       {
106     // initialize jump
107     int jump = 0;
108     // exponential distribution for load
109     std::exponential_distribution<double> exp_distribution(load_peruser);
110     // Decrement time next file arrives
111     Time_next_file_arrives -= 1;
112     // Check if the file arrived
113     if (Time_next_file_arrives <= 0) {
114         // push the time the file arrived
115         Time_file_arrived_UL.push_back(system_time);
116         // Find the next packet arrival time
117         Time_next_file_arrives = (long long) std::round(
118                 exp_distribution(generator));
119         // Push the arrived file into the buffer
120         Buffer_UL.push_back(file_size);
121         // Increment the STATISTIC total files arrived
122         total_files += 1;
123         // Increment buffer indicator
124         BUF_Ind += 1;
125         // Sum the file size to total buffer size
126         TOT_BUFF += file_size;
127     }
128     // update jump with the time next file arrives
129     jump = Time_next_file_arrives;
130 
131     if (contending == 0 && TX_ind == 0 && RX_ind == 0) {
132         // if not TX and not Contending and not RX
133         // then check the buffer and if there is data start contending
134         if (BUF_Ind > 0) {
135             // As there is data start contending
136             contending = true;
137             // set jump to zero as there is data to tx
138             jump = 0;
139             // set a uniform distribution to get values between [0 CW-1]
140             std::uniform_int_distribution<int> uni_distribution(0, CW - 1);
141             // get a random number from uniform distribution and multiply it with wifi_slot to get time in
       us
142             Counter = uni_distribution(generator) * wifi_slot;
143             // Set DIFS_Ind to zero as this is the start of DCF procedure
144             DIFS_Ind = 0;
145             // Set the DIFS Counter to DIFS duration
146             DIFS_counter = DIFS_time;
```

```
            // Initialize energy measurement to thermal noise
            Energy_meas = Noise;
            // Measure the energy from all interferers
            if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
                // initialize a vector pointer to measure PL
                std::vector<double> *PL = new std::vector<double>;
                // Measure PL from all the interferer sources
                PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
                for (auto it = PL->begin(); it != PL->end(); ++it) {
                    // Sum all the received power from all the sources where rx power is tx power*PL
                    Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
                } // End of for loop
            } // end of active devices empty condition
        } // End of buff ind>0 condition
    } // End of contending tx rx condition
    else if (contending == 1) {
        // if contending then do energy measurements
        // set jump to 0
        jump = 0;
        // Initialize energy measurement to thermal noise
        Energy_meas = Noise;
        // Measure the energy from all interferers
        if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
            // initialize a vector pointer to measure PL
            std::vector<double> *PL = new std::vector<double>;
            // Measure PL from all the interferer sources
            PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
            for (auto it = PL->begin(); it != PL->end(); ++it) {
                // Sum all the received power from all the sources where rx power is tx power*PL
                Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
            } // End of for loop
        } // end of active devices empty condition
    } // end of else if contending condition
    else {
        // might be receiving might be transmitting
        // set the jump to zero
        jump = 0;
    } // end of else
      // return time jump value
    return jump;
} // end of function check_buffer
```

3.3.3.5 void STA::check_tx_time ( )

check tx duration and decrement channel occupancy time when transmission is over notify all the listening nodes

Definition at line 279 of file STA.cpp.

```
279                         {
280     // check if STA is tranmitting
281     if (TX_ind == 1) {
282         // Decrement the channel occupy time
283         Channel_occupy_time = Channel_occupy_time - 1;
284         // check if channel occupy time is zero
285         if (Channel_occupy_time <= 0) {
286             // set tx ind to 0
287             TX_ind = 0;
288             // notify the listener group about finished transmission
289             notify_listener(FIN_TRANSMITTING);
290         } // end of chanel occupy cond
291     } // end of tx ind condition
292 } // end of function
```

3.3.3.6 void STA::jump_time (int *jump*)

used to do a time jump is all the nodes dont have data to tx

jump time if no data available at all the nodes

Definition at line 95 of file STA.cpp.

```
95                          {
96     // Substract jump time from the time next file arrives
97     Time_next_file_arrives -= jump;
98 }
```

3.3.3.7 void STA::notify_listener (int *EVENT*)

Notify the listeners This function calls the callback functions of the listeners based on the event registered INPUT: EVENT

Definition at line 685 of file STA.cpp.

```
                                     {
    switch (EVENT) {

    case TRANSMITTING:
        // Call back AP
        for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
            ((AP*) (*it))->callback_tx(this, this->TYPE);
        }

        // call back STA
        for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
            ((STA*) (*it))->callback_tx(this, this->TYPE);
        }

        break;

    case FIN_TRANSMITTING:
        // AP call back
        for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
            ((AP*) (*it))->callback_fintx(this, this->TYPE);
        }

        // STA call back
        for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
            ((STA*) (*it))->callback_fintx(this, this->TYPE);
        }

        break;
    default:
        std::cout << "unknown event occured \n";
    } // end of switch

} // end of function
```

3.3.3.8 void STA::sched_tx ( )

sched and transmit Based on the energy measuremnt from previous function Clear channel assessment is done based on CCA DCF procedures are done. If STA gets the channel then transmits

sched and transmit Based on the energy measuremnt from previous function Clear channel assessment is done based on CCA DCF procedures are done. If STA gets the channel then it schedules a UE in round robin fashion and transmits

Definition at line 195 of file STA.cpp.

```
                       {
    // DO DCF if contending
    if (contending == 1) {
        // See if DIFS is done
        if (DIFS_Ind == 1) {
            // check if the measured enrgy is less than WIFI sensing threshold
            if (Energy_meas > WIFI_TH) {
                // If energy meas id > sens thresh then bring the counter to multiple of wifi slots
                Counter = Counter - (Counter % wifi_slot) + wifi_slot;
                // Set DIFS Ind to 0
                DIFS_Ind = 0;
                // Set the DIFS counter
                DIFS_counter = DIFS_time;
            } // end of if energy condition
            else
                // Decrement the counter by 1 us If energy meas is less than sensing threshold
                Counter = Counter - 1;
            // Check whether the counter is zero
            if (Counter <= 0) {
                // If counter is zero
                // set contending to false
                contending = false;
                // Set DIFS indicator to zero
                DIFS_Ind = 0;
                // Set DIFS counter
```

```
                DIFS_counter = DIFS_time;
                // Set TX indicater to 1
                TX_ind = 1;
                // Start Scheduling
                // Set channel occupy time to zero;
                Channel_occupy_time = 0;
                // check if the buffer is non empty (Redundant condition)
                if (TOT_BUFF > 0) {
                    // increase the attempts counter
                    attempt += 1;
                    // see how much chanel time required for transmission
                    long long time = std::ceil(
                            (double) TOT_BUFF / WIFI_data_20us[MCS_UL]);
                    // check whether the time required is less than the MAX tx opportunity
                    if (20 * time < MAX_TXOP) { // 20us is the minimum granularity
                        // update the channel occupy time
                        Channel_occupy_time = 20 * time;
                        // Update Allocated data
                        ((AP*) BS_obj)->ALLOC_data_UL[ID] = TOT_BUFF;
                        // update the pumped data
                        pumped_data += TOT_BUFF;
                    } // end of if time condition
                    else {
                        // occupy the channel for max tx op
                        Channel_occupy_time = MAX_TXOP;
                        // update Allocated data
                        ((AP*) BS_obj)->ALLOC_data_UL[ID] =
                                WIFI_data_20us[MCS_UL] * MAX_TXOP / 20;
                        // update pumped data
                        pumped_data += WIFI_data_20us[MCS_UL] * MAX_TXOP / 20;
                    } // end of else
                } // end of TOT buf condition
                else {
                    // print error for debugging logical errors
                    std::cout
                            << "###### Critical Error: contending when no data ##### "
                            << std::endl;
                }
                // Notify the listening group about this transmission
                notify_listener(TRANSMITTING);
            } // end of counter zero
        } // End of DIFS ind
        else if (Energy_meas < WIFI_TH) { // if measured energy is less than the WIFI sensing threshold (1us)
            // Decrement DIFS counter by 1us
            DIFS_counter = DIFS_counter - 1;
            // Check if the DIFS counter is zero
            if (DIFS_counter <= 0) {
                // Indicate that DIFS is done
                DIFS_Ind = 1;
                // initialize the DIFS counter (Redundant but to be sure)
                DIFS_counter = DIFS_time;
            } // end of DIFS_counter
        } // end of enregy meas
    } // End of if contending
} // End of function
```

3.3.3.9 void STA::STA_init (int *ID_in,* double *coordinates_in[2],* parameters *params,* void ∗ *BS_object*)

initilaizer function Initializes the object after its creation INPUT: ID , coordinates, parameters, AP object Handle

Definition at line 36 of file STA.cpp.

```
                           {
    // Unique number to identify the AP
    ID = ID_in;
    // X coordinate
    coordinates[0] = coordinates_in[0];
    // Y coordinate
    coordinates[1] = coordinates_in[1];
    // Height of the AP from the ground
    height = params.height_AP;
    // Associated AP object Handle
    BS_obj = BS_object;
    // Minimum Contention window size of DCF
    CW_min = params.CW_min;
    // Maximum Contention window size of DCF
    CW_max = params.CW_max;
    // slot duration of DCF
    wifi_slot = params.wifi_slot;
    // DIFS duration
    DIFS_time = params.DIFS_time;
```

```
    // Maximum transmit opportunity in micro seconds
    MAX_TXOP = params.MAX_TXOP;
    // Uplink load per user in packets per micro second
    load_peruser = params.load_peruser_ul;
    // Noise floor of the receiver in Watts measured over 1 subcarrier
    Noise = params.Noise;
    // Transmit power of the AP in Watts measured over 1 subcarrier
    Pt_W = params.Pt_STA;
    // Carrier frequency used for Pathloss measurements
    fc = params.fc;
    // Channel Model
    channel_type = params.channel_model;
    // Packet size at the application
    file_size = params.file_size;
    // Wifi Energy Detect Sensing threshold
    WIFI_TH = params.WIFI_TH;
    // Maximum MCS value
    MAX_MCS = params.MAX_MCS;
    // Set the MCS (MAX) for DL transmission for all the STAs
    MCS_UL = MAX_MCS - 1;
    // bits transmitted in 20micro seconds for each MCS
    WIFI_data_20us.insert(WIFI_data_20us.begin(), &params.WIFI_data_20us[0],
            &params.WIFI_data_20us[MAX_MCS]);
    // Exponential random variable initializer
    std::exponential_distribution<double> exp_distribution(load_peruser);
    // time after which next file arrives
    Time_next_file_arrives = (long long) std::round(
            exp_distribution(generator));
    // Initialize attemp
    attempt = 0;
    // Initialize success
    success = 0;
    // Set the curtrent contention window to the minimum
    CW = CW_min;
}
```

3.3.4 Member Data Documentation

3.3.4.1 int STA::BUF_Ind = 0

Total files yet to be transmitted 0 if no data !! >0 if data counts currnt files

Definition at line 121 of file STA.h.

3.3.4.2 int STA::ID

Device ID and Device TYPE along with AP device ID will give a unique ID to the device So there is no need for a physical MAC Address Device ID

Definition at line 89 of file STA.h.

3.3.4.3 std::vector<void *> STA::listener_STA

Event listeners vector containing the station object handles

Definition at line 146 of file STA.h.

3.3.4.4 bool STA::RX_ind = 0

Reception status indicator 0 if not rx 1 if rx

Definition at line 114 of file STA.h.

3.3.4.5 bool STA::TX_ind = 0

Data allocated in uplink from each STA is available at AP class This is done inorder to rmove explicit signalling of control informationTransmission status indicator 0 if not tx 1 if tx

Definition at line 111 of file STA.h.

The documentation for this class was generated from the following files:

- /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/STA.h
- /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/STA.cpp

Chapter 4

File Documentation

4.1 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/AP.h File Reference

AP class.

```
#include "common.h"
```

Classes

- class AP

4.1.1 Detailed Description

AP class.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file AP.h.

4.2 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/BLER.h File Reference

BLER Tables for 802.11ac.

```
#include "common.h"
```

Variables

- std::vector< double > AWGN_11ac_8_MCS_BLER [8]
  BLER table for 8 MCS in WiFi.
- std::vector< double > AWGN_11ac_8_MCS_SNR [8]
  SNR corresponding to the above table.

4.2.1 Detailed Description

BLER Tables for 802.11ac.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file BLER.h.

4.2.2 Variable Documentation

4.2.2.1 std::vector<double> AWGN_11ac_8_MCS_BLER[8]

Initial value:

```
= { { 1, 0.3387, 0.0221, 0.0070,
        0.004 }, { 1, 0.5580, 0.1480, 0.0241, 0.001 }, { 0.97, 0.611, 0.168,
        0.022, 0.001 }, { 0.99, 0.845, 0.359, 0.067, 0.008 }, { 0.964, 0.53,
        0.145, 0.021, 0.001 }, { 0.953, 0.482, 0.122, 0.022, 0.003 }, { 0.776,
        0.281, 0.064, 0.0090, 0.001 }, { 0.983, 0.614, 0.177, 0.0290, 0.004 } }
```

BLER table for 8 MCS in WiFi.

Definition at line 25 of file BLER.h.

4.2.2.2 std::vector<double> AWGN_11ac_8_MCS_SNR[8]

Initial value:

```
= { { 0.7, 1.7, 2.7, 3.7, 4.7 }, {
        1.7, 2.7, 3.7, 4.7, 5.7 }, { 3.7, 4.7, 5.7, 6.7, 7.7 }, { 5.7, 6.7, 7.7,
        8.7, 9.7 }, { 9.7, 10.7, 11.7, 12.7, 13.7 }, { 13.7, 14.7, 15.7, 16.7,
        17.7 }, { 15.7, 16.7, 17.7, 18.7, 19.7 },
        { 16.7, 17.7, 18.7, 19.7, 20.7 } }
```

SNR corresponding to the above table.

Definition at line 31 of file BLER.h.

4.3 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/common.h File Reference

common APIs

```
#include <string>
#include <list>
#include <iostream>
#include <cstdlib>
#include <vector>
#include <cmath>
#include <random>
#include <algorithm>
#include "channel_models.h"
```

Classes

- struct parameters

Macros

- #define TRANSMITTING 0
- #define **FIN_TRANSMITTING** 1

Functions

- bool is_inside_rectangle (double dimension[2], double coord[2])

  checks whether the coordinates are inside a rectangle given by 'dimension[2]'
- void print_params ()

  Print the parameters.
- bool **find_prb_collision** (double SINR_db, int MCS_index)
- double Monotone_cubic_spline_interpolation (std::vector< double > *SNR_LUT, std::vector< double > *BLER_LUT, double SNR_db)

4.3.1 Detailed Description

common APIs

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file common.h.

4.3.2 Macro Definition Documentation

4.3.2.1 #define TRANSMITTING 0

Common supporting functions

Definition at line 41 of file common.h.

4.3.3 Function Documentation

4.3.3.1 double Monotone_cubic_spline_interpolation (std::vector< double > * *SNR_LUT*, std::vector< double > * *BLER_LUT*, double *SNR_db*)

Monotone_cubic_spline_interpolation https://en.wikipedia.org/wiki/Monotone_cubic_-interpolation

Definition at line 107 of file common.cpp.

```
                                                    {
    int i = 0;
    // compute dx i.e SNR differences
    std::vector<double> dx;
    for (auto it = SNR_LUT->begin(); it != SNR_LUT->end() - 1; ++it) // runs from i = 1...n-1
        {
        dx.push_back(*(it + 1) - *(it));
    }
    // compute dy i.e BLER differences
    std::vector<double> dy;
    for (auto it = BLER_LUT->begin(); it != BLER_LUT->end() - 1; ++it) // runs from i = 1...n-1
        {
        dy.push_back(*(it + 1) - *(it));
    }
    // compute slopes dy/dx
    std::vector<double> ms;
    for (auto it = dx.begin(); it != dx.end(); ++it) // runs from i = 1...n-1
        {
        ms.push_back(dy[i] / (*it));
        i++;
    }
    std::vector<double> c1s;
    std::vector<double> c2s;
    std::vector<double> c3s;

    // compute c1s first order coefficients
```

```
    c1s.push_back(ms.front());
    for (i = 0; i < ms.size() - 1; i++) {
        if (ms[i] * ms[i + 1] < 0)
            c1s.push_back(0);
        else
            c1s.push_back(
                    3 * (dx[i] + dx[i + 1])
                            / (((dx[i] + dx[i + 1] + dx[i + 1]) / ms[i])
                                    + ((dx[i] + dx[i + 1] + dx[i]) / ms[i + 1])));
    }
    c1s.push_back(ms.back());

    // compute c2s and c3s 2nd and third order coeffs
    for (i = 0; i < c1s.size() - 1; i++) {
        c2s.push_back(
                (ms[i] - c1s[i] - c1s[i] - c1s[i + 1] + ms[i] + ms[i]) / dx[i]);
        c3s.push_back((c1s[i] + c1s[i + 1] - ms[i] - ms[i]) / (dx[i] * dx[i]));
    }
    double BLER_est = 0;
    bool do_interpolation = true;

    // see if we can avoid interpolation
    for (i = 0; i < SNR_LUT->size(); i++) {
        if (*(SNR_LUT->data() + i) == SNR_db) {
            BLER_est = *(BLER_LUT->data() + i);
            do_interpolation = false;
            break;
        }
    }

    if (do_interpolation) {
        // see if the input is not in range
        if (SNR_db < SNR_LUT->front()) {
            BLER_est = BLER_LUT->front();
        } else if (SNR_db > SNR_LUT->back()) {
            BLER_est = BLER_LUT->back();
        } else {
            // start interpolation
            for (i = 0; i < SNR_LUT->size(); i++) {
                if (SNR_db < *(SNR_LUT->data() + i)) {
                    double y_low = *(BLER_LUT->data() + i - 1);
                    double x_low = *(SNR_LUT->data() + i - 1);
                    double diff = SNR_db - x_low;
                    BLER_est = y_low + (c1s[i - 1] * diff)
                            + (c2s[i - 1] * diff * diff)
                            + (c3s[i - 1] * diff * diff * diff);
                    break;
                } // end of if
            } // end of for
        } // end of else
    } // end of if
    return BLER_est;
} // end of function
```

4.4 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/STA.h File Reference

STA class.

`#include "common.h"`

Classes

- class STA

4.4.1 Detailed Description

STA class.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file STA.h.

4.5 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/channel_models/channel-_models.h File Reference

channel models

```
#include <iostream>
#include <vector>
#include <cmath>
#include <random>
```

Functions

- void PL_measure (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc, std::string channel_type)
- void InH (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)
- void **UMa** (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)
- void **UMi** (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)
- void **RMa** (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)

4.5.1 Detailed Description

channel models

Author

sreekanth dama

Date

2016

Version

0.0 IITH : sreekanth@iith.ac.in

Note

Warning

Definition in file channel_models.h.

4.5.2 Function Documentation

4.5.2.1 void InH (std::vector< double > *ACTIVE_devices*, std::vector< double > * *PL_out*, double *fc*)

Indoor Hotspot model is from TS "36.814 Table B.1.2.1-1 Summary table of the primary module path loss models" Valid for small indoor room layouts with maximum node distance of 150m Typical cell layouts 100mX100m

Definition at line 26 of file Indoor_hotspot.cpp.

```
27                      {
28     for (auto it = ACTIVE_devices.begin(); it != ACTIVE_devices.end(); ++it)
29 
30     {
31         *it = *it < 3 ? 3.1 : *it; // to make sure the distance is more than 3m
32         // Raise a warning if the cell size is more than 150m
33         if (*it > 150) {
34             std::cout
35                     << "_#_#_#_WARNING_#_#_#: d>150 \n ### InH MODEL applicable only for cells of size < 150m ###"
36                     << std::endl;
37         }
38         // probability of LOS
39         double Pr_LOS = 0;
40         if (*it <= 18)
41             Pr_LOS = 1;
42         else if (*it > 18 && *it < 37)
43             Pr_LOS = std::exp(-(*it - 18) / 27);
44         else if (*it >= 37)
45             Pr_LOS = 0.5;
46         // LOS is binomial random variable taking 0(NLOS) or 1(LOS)
47         std::binomial_distribution<int> bi_distribution(1, Pr_LOS);
48         // random device uses the system entropy
49         std::random_device generator;
50         // Calculate LOS
51         int LOS = bi_distribution(generator);
52         // LOS is applicable only in the range 3m-100m
53         if (*it > 100) {
54             LOS = 0;
55         }
56         // Path loss in dB
57         double PL = 0;
58         if (LOS == 1) {
59             PL = 16.9 * std::log10(*it) + 32.8 + 20 * std::log10(fc);
60         } else {
61             PL = 43.3 * std::log10(*it) + 11.5 + 20 * std::log10(fc);
62         }
63         // Shadowing Model is lognormal
64         double PL_shd = 0;
65         if (LOS == 1) {
66             // Gaussian distribution with mean 0 and variance 3 for LOS
67             std::normal_distribution<double> norm_distribution(0, 3.0);
68             PL_shd = norm_distribution(generator);
69         } else {
70             // Gaussian distribution with mean 0 and variance 4 for NLOS
71             std::normal_distribution<double> norm_distribution(0, 4.0);
72             PL_shd = norm_distribution(generator);
73         }
74 
75         PL_out->push_back(-PL + PL_shd);
76     }
77 }
```

4.5.2.2 void PL_measure (std::vector< double > *ACTIVE_devices*, std::vector< double > ∗ *PL*, double *fc*, std::string *channel_type*)

Pl measure function INPUT: active devices, empty vector, frequency in GHz, channel model Output: PL

Definition at line 27 of file PL_measure.cpp.

```
                                            {

    if (channel_type == "InH") {
        // Indoor Hotspot Model
        InH(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "RMa") {
        // Rural Macro
        RMa(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "UMa") {
        // Urban Macro
        UMa(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "UMi") {
        // Urban Micro
        UMi(ACTIVE_devices, PL, fc);
    }
}
```

4.6 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/AP.cpp File Reference

AP class source.

```
#include "common.h"
#include "AP.h"
#include "STA.h"
```

Variables

- long long **system_time**
- long long **total_tti**

4.6.1 Detailed Description

AP class source.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file AP.cpp.

4.7 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/common.cpp File Reference

common source

```
#include "common.h"
#include "BLER.h"
```

Functions

- void print_params ()

  Print the parameters.
- bool is_inside_rectangle (double dimension[2], double coord[2])

  checks whether the coordinates are inside a rectangle given by 'dimension[2]'
- bool **find_prb_collision** (double SINR_db, int MCS_index)
- double Monotone_cubic_spline_interpolation (std::vector< double > *SNR_LUT, std::vector< double > *BLER_LUT, double SNR_db)

Variables

- parameters common_params

  Sets the simulation parameters.
- std::vector< double > AWGN_11ac_8_MCS_BLER [8]

  BLER table for 8 MCS in WiFi.
- std::vector< double > AWGN_11ac_8_MCS_SNR [8]

  SNR corresponding to the above table.

4.7.1 Detailed Description

common source

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file common.cpp.

4.7.2 Function Documentation

4.7.2.1 double Monotone_cubic_spline_interpolation (std::vector< double > * *SNR_LUT*, std::vector< double > * *BLER_LUT*, double *SNR_db*)

Monotone_cubic_spline_interpolation https://en.wikipedia.org/wiki/Monotone_cubic_interpolation

Definition at line 107 of file common.cpp.

```
                                                    {
    int i = 0;
    // compute dx i.e SNR differences
    std::vector<double> dx;
    for (auto it = SNR_LUT->begin(); it != SNR_LUT->end() - 1; ++it) // runs from i = 1...n-1
            {
        dx.push_back(*(it + 1) - *(it));
    }
    // compute dy i.e BLER differences
    std::vector<double> dy;
    for (auto it = BLER_LUT->begin(); it != BLER_LUT->end() - 1; ++it) // runs from i = 1...n-1
            {
        dy.push_back(*(it + 1) - *(it));
    }
    // compute slopes dy/dx
    std::vector<double> ms;
    for (auto it = dx.begin(); it != dx.end(); ++it) // runs from i = 1...n-1
            {
        ms.push_back(dy[i] / (*it));
        i++;
    }
    std::vector<double> c1s;
    std::vector<double> c2s;
    std::vector<double> c3s;

    // compute c1s first order coefficients
    c1s.push_back(ms.front());
    for (i = 0; i < ms.size() - 1; i++) {
        if (ms[i] * ms[i + 1] < 0)
            c1s.push_back(0);
        else
            c1s.push_back(
                    3 * (dx[i] + dx[i + 1])
                            / (((dx[i] + dx[i + 1] + dx[i + 1]) / ms[i])
                                    + ((dx[i] + dx[i + 1] + dx[i]) / ms[i + 1])));
    }
    c1s.push_back(ms.back());

    // compute c2s and c3s 2nd and third order coeffs
    for (i = 0; i < c1s.size() - 1; i++) {
        c2s.push_back(
                (ms[i] - c1s[i] - c1s[i] - c1s[i + 1] + ms[i] + ms[i]) / dx[i]);
        c3s.push_back((c1s[i] + c1s[i + 1] - ms[i] - ms[i]) / (dx[i] * dx[i]));
    }
    double BLER_est = 0;
    bool do_interpolation = true;

    // see if we can avoid interpolation
    for (i = 0; i < SNR_LUT->size(); i++) {
        if (*(SNR_LUT->data() + i) == SNR_db) {
            BLER_est = *(BLER_LUT->data() + i);
            do_interpolation = false;
            break;
        }
    }

    if (do_interpolation) {
        // see if the input is not in range
        if (SNR_db < SNR_LUT->front()) {
            BLER_est = BLER_LUT->front();
        } else if (SNR_db > SNR_LUT->back()) {
            BLER_est = BLER_LUT->back();
        } else {
```

```
            // start interpolation
            for (i = 0; i < SNR_LUT->size(); i++) {
                if (SNR_db < *(SNR_LUT->data() + i)) {
                    double y_low = *(BLER_LUT->data() + i - 1);
                    double x_low = *(SNR_LUT->data() + i - 1);
                    double diff = SNR_db - x_low;
                    BLER_est = y_low + (c1s[i - 1] * diff)
                            + (c2s[i - 1] * diff * diff)
                            + (c3s[i - 1] * diff * diff * diff);
                    break;
                } // end of if
            } // end of for
        } // end of else
    } // end of if
    return BLER_est;
} // end of function
```

4.8 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/STA.cpp File Reference

STA class source.

```
#include "common.h"
#include "STA.h"
#include "AP.h"
```

Variables

- long long **system_time**
- long long **total_tti**

4.8.1 Detailed Description

STA class source.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file STA.cpp.

4.9 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Indoor_hotspot.cpp File Reference

Indoor hotspot Model.

`#include "channel_models.h"`

Functions

- void InH (std::vector< double > ACTIVE_devices, std::vector< double > *PL_out, double fc)

4.9.1 Detailed Description

Indoor hotspot Model.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file Indoor_hotspot.cpp.

4.9.2 Function Documentation

4.9.2.1 void InH (std::vector< double > *ACTIVE_devices*, std::vector< double > * *PL_out*, double *fc*)

Indoor Hotspot model is from TS "36.814 Table B.1.2.1-1 Summary table of the primary module path loss models" Valid for small indoor room layouts with maximum node distance of 150m Typical cell layouts 100mX100m

Definition at line 26 of file Indoor_hotspot.cpp.

```
                    {
    for (auto it = ACTIVE_devices.begin(); it != ACTIVE_devices.end(); ++it)

    {
        *it = *it < 3 ? 3.1 : *it; // to make sure the distance is more than 3m
        // Raise a warning if the cell size is more than 150m
        if (*it > 150) {
            std::cout
                    << "_#_#_#_WARNING_#_#_#: d>150 \n ### InH MODEL applicable only for cells of size <
    150m ###"
                    << std::endl;
        }
```

```
        // probability of LOS
        double Pr_LOS = 0;
        if (*it <= 18)
            Pr_LOS = 1;
        else if (*it > 18 && *it < 37)
            Pr_LOS = std::exp(-(*it - 18) / 27);
        else if (*it >= 37)
            Pr_LOS = 0.5;
        // LOS is binomial random variable taking 0(NLOS) or 1(LOS)
        std::binomial_distribution<int> bi_distribution(1, Pr_LOS);
        // random device uses the system entropy
        std::random_device generator;
        // Calculate LOS
        int LOS = bi_distribution(generator);
        // LOS is applicable only in the range 3m-100m
        if (*it > 100) {
            LOS = 0;
        }
        // Path loss in dB
        double PL = 0;
        if (LOS == 1) {
            PL = 16.9 * std::log10(*it) + 32.8 + 20 * std::log10(fc);
        } else {
            PL = 43.3 * std::log10(*it) + 11.5 + 20 * std::log10(fc);
        }
        // Shadowing Model is lognormal
        double PL_shd = 0;
        if (LOS == 1) {
            // Gaussian distribution with mean 0 and variance 3 for LOS
            std::normal_distribution<double> norm_distribution(0, 3.0);
            PL_shd = norm_distribution(generator);
        } else {
            // Gaussian distribution with mean 0 and variance 4 for NLOS
            std::normal_distribution<double> norm_distribution(0, 4.0);
            PL_shd = norm_distribution(generator);
        }

        PL_out->push_back(-PL + PL_shd);
    }
}
```

4.10 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/PL_measure.cpp File Reference

ALL PL Models.

`#include "channel_models.h"`

Functions

- void PL_measure (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc, std::string channel_type)

4.10.1 Detailed Description

ALL PL Models.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file PL_measure.cpp.

4.10.2 Function Documentation

4.10.2.1 void PL_measure (std::vector< double > *ACTIVE_devices,* std::vector< double > ∗ *PL,* double *fc,* std::string *channel_type*)

Pl measure function INPUT: active devices, empty vector, frequency in GHz, channel model Output: PL

Definition at line 27 of file PL_measure.cpp.

```
                                           {

    if (channel_type == "InH") {
        // Indoor Hotspot Model
        InH(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "RMa") {
        // Rural Macro
        RMa(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "UMa") {
        // Urban Macro
        UMa(ACTIVE_devices, PL, fc);
    }
    if (channel_type == "UMi") {
        // Urban Micro
        UMi(ACTIVE_devices, PL, fc);
    }
}
```

4.11 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/-Rural_Macro.cpp File Reference

Rural Macro Model.

`#include "channel_models.h"`

Functions

- void **RMa** (std::vector< double > ACTIVE_devices, std::vector< double > ∗PL, double fc)

4.11.1 Detailed Description

Rural Macro Model.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file Rural_Macro.cpp.

4.12 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/-Urban_Macro.cpp File Reference

Urban Macro Model.

```
#include "channel_models.h"
```

Functions

- void **UMa** (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)

4.12.1 Detailed Description

Urban Macro Model.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file Urban_Macro.cpp.

4.13 /home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/-Urban_Micro.cpp File Reference

Urban Micro Model.

`#include "channel_models.h"`

Functions

- void **UMi** (std::vector< double > ACTIVE_devices, std::vector< double > *PL, double fc)

4.13.1 Detailed Description

Urban Micro Model.

Author

sreekanth dama

Date

2016

Version

0.0

Note

Warning

Definition in file Urban_Micro.cpp.