

HetNetSim

Generated by Doxygen 1.8.6

Mon Oct 10 2016 19:17:00



# Contents

<b>1</b>	<b>Class Index</b>	<b>1</b>
1.1	Class List . . . . .	1
<b>2</b>	<b>File Index</b>	<b>3</b>
2.1	File List . . . . .	3
<b>3</b>	<b>Class Documentation</b>	<b>5</b>
3.1	AP Class Reference . . . . .	5
3.1.1	Detailed Description . . . . .	6
3.1.2	Constructor & Destructor Documentation . . . . .	7
3.1.2.1	AP . . . . .	7
3.1.3	Member Function Documentation . . . . .	7
3.1.3.1	add_listener . . . . .	7
3.1.3.2	ADD_STA . . . . .	7
3.1.3.3	AP_init . . . . .	7
3.1.3.4	callback_fintx . . . . .	8
3.1.3.5	callback_tx . . . . .	12
3.1.3.6	check_buffer . . . . .	14
3.1.3.7	check_tx_time . . . . .	15
3.1.3.8	jump_time . . . . .	15
3.1.3.9	notify_listener . . . . .	15
3.1.3.10	sched_tx . . . . .	16
3.1.4	Member Data Documentation . . . . .	17
3.1.4.1	ALLOC_data_DL . . . . .	17
3.1.4.2	ALLOC_data_UL . . . . .	17
3.1.4.3	BUF_Ind . . . . .	17
3.1.4.4	ID . . . . .	18
3.1.4.5	listener_STA . . . . .	18
3.1.4.6	RX_ind . . . . .	18
3.1.4.7	TX_ind . . . . .	18
3.2	parameters Struct Reference . . . . .	18
3.2.1	Detailed Description . . . . .	19

3.2.2	Member Data Documentation . . . . .	19
3.2.2.1	CW_min . . . . .	19
3.2.2.2	DL_UL . . . . .	19
3.2.2.3	NCell . . . . .	20
3.3	STA Class Reference . . . . .	20
3.3.1	Detailed Description . . . . .	21
3.3.2	Constructor & Destructor Documentation . . . . .	21
3.3.2.1	STA . . . . .	21
3.3.3	Member Function Documentation . . . . .	22
3.3.3.1	add_listener . . . . .	22
3.3.3.2	callback_fintx . . . . .	22
3.3.3.3	callback_tx . . . . .	26
3.3.3.4	check_buffer . . . . .	27
3.3.3.5	check_tx_time . . . . .	28
3.3.3.6	jump_time . . . . .	28
3.3.3.7	notify_listener . . . . .	29
3.3.3.8	sched_tx . . . . .	29
3.3.3.9	STA_init . . . . .	30
3.3.4	Member Data Documentation . . . . .	31
3.3.4.1	BUF_Ind . . . . .	31
3.3.4.2	ID . . . . .	31
3.3.4.3	listener_STA . . . . .	31
3.3.4.4	RX_ind . . . . .	31
3.3.4.5	TX_ind . . . . .	31
<b>4</b>	<b>File Documentation</b>	<b>33</b>
4.1	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/AP.h File Reference . . . . .	33
4.1.1	Detailed Description . . . . .	33
4.2	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/BLER.h File Reference . . . . .	34
4.2.1	Detailed Description . . . . .	34
4.2.2	Variable Documentation . . . . .	34
4.2.2.1	AWGN_11ac_8_MCS_BLER . . . . .	34
4.2.2.2	AWGN_11ac_8_MCS_SNR . . . . .	35
4.3	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/common.h File Reference . . . . .	35
4.3.1	Detailed Description . . . . .	35
4.3.2	Macro Definition Documentation . . . . .	36
4.3.2.1	TRANSMITTING . . . . .	36
4.3.3	Function Documentation . . . . .	36
4.3.3.1	Monotone_cubic_spline_interpolation . . . . .	36
4.4	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/STA.h File Reference . . . . .	37

4.4.1	Detailed Description	37
4.5	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/channel_models/channel_models.h File Reference	38
4.5.1	Detailed Description	38
4.5.2	Function Documentation	39
4.5.2.1	InH	39
4.5.2.2	PL_measure	40
4.6	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/AP.cpp File Reference	40
4.6.1	Detailed Description	40
4.7	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/common.cpp File Reference	41
4.7.1	Detailed Description	41
4.7.2	Function Documentation	42
4.7.2.1	Monotone_cubic_spline_interpolation	42
4.8	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/STA.cpp File Reference	43
4.8.1	Detailed Description	43
4.9	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Indoor_hotspot.cpp File Reference	44
4.9.1	Detailed Description	44
4.9.2	Function Documentation	44
4.9.2.1	InH	44
4.10	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/PL_measure.cpp File Reference	45
4.10.1	Detailed Description	45
4.10.2	Function Documentation	46
4.10.2.1	PL_measure	46
4.11	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Rural_Macro.cpp File Reference	46
4.11.1	Detailed Description	46
4.12	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Urban_Macro.cpp File Reference	47
4.12.1	Detailed Description	47
4.13	/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Urban_Micro.cpp File Reference	48
4.13.1	Detailed Description	48



# Chapter 1

## Class Index

### 1.1 Class List

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

<a href="#">AP</a> . . . . .	5
<a href="#">parameters</a> . . . . .	18
<a href="#">STA</a> . . . . .	20





## Chapter 2

# File Index

### 2.1 File List

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

/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/AP.h	
AP class	33
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/BLER.h	
BLER Tables for 802.11ac	34
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/common.h	
Common APIs	35
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/802.11ac/STA.h	
STA class	37
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/hdr/channel_models/channel_models.h	
Channel models	38
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/AP.cpp	
AP class source	40
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/common.cpp	
Common source	41
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/802.11ac/STA.cpp	
STA class source	43
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Indoor_hotspot.cpp	
Indoor hotspot Model	44
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/PL_measure.cpp	
ALL PL Models	45
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Rural_Macro.cpp	
Rural Macro Model	46
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Urban_Macro.cpp	
Urban Macro Model	47
/home/sreekanthepc/git_bitbucket_repos/hetnetsim_cpp/src/channel_models/Urban_Micro.cpp	
Urban Micro Model	48



## 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 tranmisted 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;
```

```

48 // Maximum Contention window size of DCF
49 CW_max = params.CW_max;
50 // slot duration of DCF
51 wifi_slot = params.wifi_slot;
52 // DIFS duration
53 DIFS_time = params.DIFS_time;
54 // Maximum transmit opportunity in micro seconds
55 MAX_TXOP = params.MAX_TXOP;
56 // Downlink load per user in packets per micro second
57 load_peruser = params.load_peruser_dl;
58 // Total number of STAs associated to this AP
59 Connected_UEs = params.n_wifi;
60 // Noise floor of the receiver in Watts measured over 1 subcarrier
61 Noise = params.Noise;
62 // Transmit power of the AP in Watts measured over 1 subcarrier
63 Pt_W = params.Pt_AP;
64 // Carrier frequency used for Pathloss measurements
65 fc = params.fc;
66 // Channel Model
67 channel_type = params.channel_model;
68 // Packet size at the application
69 file_size = params.file_size;
70 // Wifi Energy Detect Sensing threshold
71 WIFI_TH = params.WIFI_TH;
72 // Maximum MCS value
73 MAX_MCS = params.MAX_MCS;
74 // Set the MCS (MAX) for DL transmission for all the STAs
75 MCS_DL.insert(MCS_DL.begin(), Connected_UEs, MAX_MCS - 1);
76 // bits transmitted in 20micro seconds for each MCS
77 WIFI_data_20us.insert(WIFI_data_20us.begin(), &params.WIFI_data_20us[0],
78 &params.WIFI_data_20us[MAX_MCS]);
79 /*! Data allocated in downlink for each STA
80 * This is necessary as there is no explicit signalling of control information
81 */
82 ALLOC_data_DL.insert(ALLOC_data_DL.begin(), Connected_UEs, 0);
83 /*! Data allocated in uplink from each STA
84 * This is necessary as there is no explicit signalling of control information
85 */
86 ALLOC_data_UL.insert(ALLOC_data_UL.begin(), Connected_UEs, 0);
87 // Exponential random variable initializer
88 std::exponential_distribution<double> exp_distribution(load_peruser);
89 // Setting up the buffers and initializing the time first packet arrives
90 for (int i = 0; i < Connected_UEs; i++) {
91 // create a new linked list for Data buffering for each STA
92 std::list<long long> *temp0 = new std::list<long long>;
93 // push the list into the Buffer vector
94 Buffer_DL.push_back(*temp0);
95 // create a new linked list for time file arrived for each STA
96 std::list<long long> *temp1 = new std::list<long long>;
97 // push the list into the Time file arrived vector
98 Time_file_arrived_DL.push_back(*temp1);
99 // Find the first packet arrival for a user in DL
100 int time_temp = (long long) std::round(exp_distribution(generator));
101 // push the time first packet arrives to the vector
102 Time_next_file_arrives.push_back(time_temp);
103 }
104 // Initialize attempt
105 attempt.insert(attempt.begin(), Connected_UEs, 0);
106 // Initialize success
107 success.insert(success.begin(), Connected_UEs, 0);
108 // initializ the sum of data for all STA to zero.
109 SUM_BUFF_DL.insert(SUM_BUFF_DL.begin(), Connected_UEs, 0);
110 // Set the current contention window to the minimum
111 CW = CW_min;
112 }

```

### 3.1.3.4 void AP::callback\_fintx ( void \* obj, std::string type )

Callback function for finished tx When a device finishes transmission and notifies 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](#) then 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
    // stop receiving
    RX_ind = 0;
    // 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(
    coordinates[0],
    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)));
    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
    ((STA*) obj)->SINR.push_back(SINR_db);
    // find if the transmission is in collision based on the SINR
    bool collision = find_prb_collision(SINR_db, ((STA*) obj)->MCS_UL);
    // update global variable total_tti marking finish of 1 tti
    total_tti++;
    if (collision == 0) {
    // if packet not in collision
    // update the success
    ((STA*) obj)->success = ((STA*) obj)->success + 1;
    // Decrement the data received from the UL buffer
    ((STA*) obj)->Buffer_UL.front() -= ALLOC_data_UL[((
STA*) obj)->ID];
    // increment the served data STAT
    ((STA*) obj)->served_data += ALLOC_data_UL[((STA*) obj)->ID];
    // decrement the total buff
    ((STA*) obj)->TOT_BUFF -= ALLOC_data_UL[((STA*) obj)->ID];
    // if the file is finished
    if (((STA*) obj)->Buffer_UL.front() <= 0) {
    // record the time taken for the file to download
    int te2 = system_time
    - ((STA*) obj)->Time_file_arrived_UL.front();
    // push the time taken for STATs
    ((STA*) obj)->TTTDeliverOfFile_UL.push_back(te2);
    // pop the time the file arrived
    ((STA*) obj)->Time_file_arrived_UL.pop_front();
    // see if there is another file and check if that is scheduled too
    if (((STA*) obj)->TOT_BUFF > 0) {
    // check the top of the buffer

```

```

533         int temp = ((STA*) obj)->Buffer_UL.front();
534         // pop it out of the buffer
535         ((STA*) obj)->Buffer_UL.pop_front();
536         // ADD it to the next packet
537         ((STA*) obj)->Buffer_UL.front() += temp;
538     } else {
539         // due 20us granularity some zero padding will happen
540         // if there is no new packet remove the zeros from counting into served data
541         ((STA*) obj)->served_data += ((STA*) obj)->TOT_BUFF;
542         // remove the zero padding
543         ((STA*) obj)->TOT_BUFF = 0;
544         // remove the finished packet from buffer
545         ((STA*) obj)->Buffer_UL.pop_front();
546     } // end of else
547     // decrement that file from the BUF_Ind
548     ((STA*) obj)->BUF_Ind = ((STA*) obj)->BUF_Ind - 1;
549 } // end of buffer_ul
550 // MCS Adaptation
551 // if total attempts for this STA are 100
552 if (((STA*) obj)->attempt == 100) {
553     // check among the 100 attempts >70 are success
554     if (((STA*) obj)->success > 70) {
555         // then increase the MCS
556         ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
557         // if the MCS is >= MAX MCS
558         if (((STA*) obj)->MCS_UL >= MAX_MCS)
559             // then set it to MAX MCS
560             ((STA*) obj)->MCS_UL = MAX_MCS - 1;
561     } // end of success
562     // check if the success are less than 30%
563     else if (((STA*) obj)->success < 30) {
564         // then decrement the MCS
565         ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
566         // if the MCS is less than the minimum
567         if (((STA*) obj)->MCS_UL < 0)
568             // then set it to minimum
569             ((STA*) obj)->MCS_UL = 0;
570     } // end of if else
571     // set the attempts to 0 for this STA
572     ((STA*) obj)->attempt = 0;
573     // set the success to 0 for this STA
574     ((STA*) obj)->success = 0;
575 } // end of attempt condition
576 // update contention window to the minimum
577 ((STA*) obj)->CW = CW_min;
578 // reset ALLOC UL
579 ALLOC_data_UL[((STA*) obj)->ID] = 0;
580 } // end of collision condition
581 // if it is collision
582 else {
583     // double the contention window
584     ((STA*) obj)->CW = ((STA*) obj)->CW * 2;
585     // if the current window is maximum or greater
586     if (((STA*) obj)->CW > CW_max)
587         // set it to maximum
588         ((STA*) obj)->CW = CW_max;
589     // MCS Adaptation
590     // if total attempts for this STA are 100
591     if (((STA*) obj)->attempt == 100) {
592         // check among the 100 attempts >70 are success
593         if (((STA*) obj)->success > 70) {
594             // then increase the MCS
595             ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
596             // if the MCS is >= MAX MCS
597             if (((STA*) obj)->MCS_UL >= MAX_MCS)
598                 // then set it to MAX MCS
599                 ((STA*) obj)->MCS_UL = MAX_MCS - 1;
600         } // end of success
601         // check if the success are less than 30%
602         else if (((STA*) obj)->success < 30) {
603             // then decrement the MCS
604             ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
605             // if the MCS is less than the minimum
606             if (((STA*) obj)->MCS_UL < 0)
607                 // then set it to minimum
608                 ((STA*) obj)->MCS_UL = 0;
609         } // end of if else
610         // set the attempts to 0 for this STA
611         ((STA*) obj)->attempt = 0;
612         // set the success to 0 for this STA
613         ((STA*) obj)->success = 0;
614     } // end of attempt condition
615 }
616 // reset ALLOC UL
617 ALLOC_data_UL[((STA*) obj)->ID] = 0;
618 } // end of if temp
619 // if EVENT from associated STA and not receiving this STA

```



```

620     else if (temp) {
621         // double the contention window
622         ((STA*) obj)->CW = ((STA*) obj)->CW * 2;
623         // if the current window is maximum or greater
624         if (((STA*) obj)->CW > CW_max)
625             // set it to maximum
626             ((STA*) obj)->CW = CW_max;
627         // MCS Adaptation
628         // if total attempts for this STA are 100
629         if (((STA*) obj)->attempt == 100) {
630             // check among the 100 attempts >70 are success
631             if (((STA*) obj)->success > 70) {
632                 // then increase the MCS
633                 ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL + 1;
634                 // if the MCS is >= MAX MCS
635                 if (((STA*) obj)->MCS_UL >= MAX_MCS)
636                     // then set it to MAX MCS
637                     ((STA*) obj)->MCS_UL = MAX_MCS - 1;
638             } // end of success
639             // check if the success are less than 30%
640             else if (((STA*) obj)->success < 30) {
641                 // then decrement the MCS
642                 ((STA*) obj)->MCS_UL = ((STA*) obj)->MCS_UL - 1;
643                 // if the MCS is less than the minimum
644                 if (((STA*) obj)->MCS_UL < 0)
645                     // then set it to minimum
646                     ((STA*) obj)->MCS_UL = 0;
647             } // end of if else
648             // set the attempts to 0 for this STA
649             ((STA*) obj)->attempt = 0;
650             // set the success to 0 for this STA
651             ((STA*) obj)->success = 0;
652         } // end of attempt condition
653         // reset ALLOC UL
654         ALLOC_data_UL[((STA*) obj)->ID] = 0;
655         // find the distance from the AP
656         double d = std::sqrt(
657             std::pow(coordinates[0] - ((STA*) obj)->
coordinates[0], 2)
658             + std::pow(
659                 coordinates[1] - ((STA*) obj)->
coordinates[1],
660                 2)
661             + std::pow(height - ((STA*) obj)->height, 2));
662         if (RX_ind == 1) {
663             // Measure the PL
664             double PL_meas = 0;
665             if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
666                 // initialize a vector pointer to measure PL
667                 std::vector<double> *PL = new std::vector<double>;
668                 // Measure PL from all the interferer sources
669                 PL_measure(ACTIVE_Devices, PL, fc, channel_type);
670                 for (auto it = PL->begin(); it != PL->end(); ++it) {
671                     // Sum all the received power from all the sources where rx power is tx power*PL
672                     PL_meas += Pt_W * std::pow(10, *it / 10);
673                 }
674             } // end of active device
675             // Add the interferece power of previous state
676             Int_power += PL_meas * (system_time - local_time);
677             // record the time the interference changed
678             local_time = system_time;
679             // remove the STA from the interferers list
680             for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
681                 ++it) {
682                 if (*it == d) {
683                     ACTIVE_Devices.erase(it);
684                     break;
685                 }
686             } // end of for
687         } // end of if
688         else {
689             // remove the STA from the interferers list
690             for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
691                 ++it) {
692                 if (*it == d) {
693                     ACTIVE_Devices.erase(it);
694                     break;
695                 }
696             } // end of for
697         } // end of else
698     } // end of elseif
699     else {
700         double d = 0;
701         if (type == "STA")
702             d = std::sqrt(

```

```

703         std::pow(coordinates[0] - ((STA*) obj)->
coordinates[0], 2)
704             + std::pow(
705                 coordinates[1]
706                 - ((STA*) obj)->coordinates[1], 2)
707             + std::pow(height - ((STA*) obj)->height, 2));
708     if (type == "AP")
709         d = std::sqrt(
710             std::pow(coordinates[0] - ((AP*) obj)->
coordinates[0], 2)
711             + std::pow(
712                 coordinates[1]
713                 - ((AP*) obj)->coordinates[1], 2)
714             + std::pow(height - ((AP*) obj)->height, 2));
715
716     if (RX_ind == 1) {
717         // Measure the PL
718         double PL_meas = 0;
719         if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
720             // initialize a vector pointer to measure PL
721             std::vector<double> *PL = new std::vector<double>;
722             // Measure PL from all the interferer sources
723             PL_measure(ACTIVE_Devices, PL, fc, channel_type);
724             for (auto it = PL->begin(); it != PL->end(); ++it) {
725                 // Sum all the received power from all the sources where rx power is tx power*PL
726                 PL_meas += Pt_W * std::pow(10, *it / 10);
727             }
728         } // end of active device
729         // Add the interferece power of previous state
730         Int_power += PL_meas * (system_time - local_time);
731         // record the time the interference changed
732         local_time = system_time;
733         // remove the device from the interferers list
734         for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
735             ++it) {
736             if (*it == d) {
737                 ACTIVE_Devices.erase(it);
738                 break;
739             }
740         } // end of for
741     } else {
742         // remove the device from the interferers list
743         for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
744             ++it) {
745             if (*it == d) {
746                 ACTIVE_Devices.erase(it);
747                 break;
748             }
749         }
750     }
751 }
752 }

```

### 3.1.3.5 void AP::callback\_tx ( void \* obj, std::string type )

#### call backs

Callback function for tx When a device transmits and notfys 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;

```

```

364
365     if (type == "AP")
366         // is EVENT from AP
367         temp = 0;
368     if (type == "STA") {
369         // check if the STA is associated with AP
370         for (auto it = BS_UEs.begin(); it != BS_UEs.end(); ++it) {
371             if (obj == *it) {
372                 temp = 1;
373                 break;
374             }
375         } // end of for
376     } // end if
377     if (temp && TX_ind != 1 && RX_ind != 1) {
378         // if the EVENT is from connected STA and AP is not receiving not transmitting
379         // Start receiving
380         RX_ind = 1;
381         // stop contending
382         contending = 0;
383         // record the STA ID
384         CURRENT_STA_UL = ((STA*) obj)->ID;
385         // Mark the time the transmission started
386         local_time = system_time;
387         // init sig power
388         Sig_power = 0;
389         // init interferer power
390         Int_power = 0;
391         // Record the channel occupy time
392         Channel_occupy_time = ((STA*) obj)->Channel_occupy_time;
393     } // end of temp
394     else { // else add the event sources to interferers list
395         // if the event is from other STA
396         double d = 0;
397         if (type == "STA")
398             // find the distance from the interferer
399             d = std::sqrt(
400                 std::pow(coordinates[0] - ((STA*) obj)->
401                     coordinates[0], 2)
402                     + std::pow(
403                         coordinates[1]
404                         - ((STA*) obj)->coordinates[1], 2)
405                     + std::pow(height - ((STA*) obj)->height, 2));
406         if (type == "AP")
407             // measure distance from AP
408             d = std::sqrt(
409                 std::pow(coordinates[0] - ((AP*) obj)->
410                     coordinates[0], 2)
411                     + std::pow(
412                         coordinates[1]
413                         - ((AP*) obj)->coordinates[1], 2)
414                     + std::pow(height - ((AP*) obj)->height, 2));
415         // check if the current AP is in receiving mode
416         if (RX_ind == 1) {
417             // Measure the PL
418             double PL_meas = 0;
419             if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
420                 // initialize a vector pointer to measure PL
421                 std::vector<double> *PL = new std::vector<double>;
422                 // Measure PL from all the interferer sources
423                 PL_measure(ACTIVE_Devices, PL, fc, channel_type);
424                 for (auto it = PL->begin(); it != PL->end(); ++it) {
425                     // Sum all the received power from all the sources where rx power is tx power*PL
426                     PL_meas += Pt_W * std::pow(10, *it / 10);
427                 }
428             } // end of active devices
429             // Add the interferece power of previous state
430             Int_power += PL_meas * (system_time - local_time);
431             // record the time the interference changed
432             local_time = system_time;
433             // push the interferer distance to interferers list
434             ACTIVE_Devices.push_back(d);
435         } // end of RX condition
436         else {
437             // Add the new interferer to the interferers list
438             ACTIVE_Devices.push_back(d);
439         } // end of else
440     } // 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
183             us 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 transmitting
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 channel 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         // Subtract 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         case TRANSMITTING:
776             // Call back AP
777             for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
778                 ((AP*) (*it))->callback_tx(this, this->TYPE);
779             }
780         }
781         // call back STA

```

```

783         for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
784             ((STA*) (*it))->callback_tx(this, this->TYPE);
785         }
786         break;
787     case FIN_TRANSMITTING:
788         // AP call back
789         for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
790             ((AP*) (*it))->callback_fintx(this, this->TYPE);
791         }
792         // STA call back
793         for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
794             ((STA*) (*it))->callback_fintx(this, this->TYPE);
795         }
796         break;
797     default:
798         std::cout << "unknown event occurred \n";
799     } // end of switch
800 } // 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 indicator 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
269                     the next one
270                     CURRENT_STA_DL = 0; // pick the first UE
271                 } else {
272                     CURRENT_STA_DL += 1; // pick the next UE
273                 } // end of else
274                 // schedule the UE
275                 while (Channel_occupy_time == 0) {
276                     // check if the current UE has data
277                     if (SUM_BUFF_DL[CURRENT_STA_DL] > 0) {
278                         // increment the attempt
279                         attempt[CURRENT_STA_DL] = attempt[
280                             CURRENT_STA_DL] + 1;
281                         // check the time required
282                         long long time =
283                             std::ceil(
284                                 (double) SUM_BUFF_DL[

```

```

CURRENT_STA_DL]
283                                     / WIFI_data_20us[MCS_DL[
CURRENT_STA_DL]]);
284                                     // check whether the time required is less than the MAX tx opportunity
285                                     if (20 * time < MAX_TXOP) { // 20us is the minimum granularity
286                                         // update the channel occupy time
287                                         Channel_occupy_time = 20 * time;
288                                         // Update Allocated data
289                                         ALLOC_data_DL[CURRENT_STA_DL] =
290                                             SUM_BUFF_DL[CURRENT_STA_DL];
291                                         // update the pumped data
292                                         pumped_data += SUM_BUFF_DL[
CURRENT_STA_DL];
293                                     } else {
294                                         // occupy the channel for max tx op
295                                         Channel_occupy_time = MAX_TXOP;
296                                         // update Allocated data
297                                         ALLOC_data_DL[CURRENT_STA_DL] =
298                                             WIFI_data_20us[MCS_DL[CURRENT_STA_DL]]
299                                                 * MAX_TXOP / 20;
300                                         // update pumped data
301                                         pumped_data +=
302                                             WIFI_data_20us[MCS_DL[CURRENT_STA_DL]]
303                                                 * MAX_TXOP / 20;
304                                     } // end of else
305                                 } else {
306                                     // Do round Robin
307                                     // if current UE is last one pick the next one
308                                     if (CURRENT_STA_DL == Connected_UEs - 1)
309                                         CURRENT_STA_DL = 0; // pick the first UE
310                                     else
311                                         CURRENT_STA_DL += 1; // pick the next UE
312                                 } // end of else
313                             } // end of inf while
314                             // Notify the listening group about this transmission
315                             notify_listener(TRANSMITTING);
316                             } // end of counter zero
317                         } else if (Energy_meas < WIFI_TH) { // if measured energy is less than the WIFI sensing threshold
(lus)
318                             // Decrement DIFS counter by lus
319                             DIFS_counter = DIFS_counter - 1;
320                             // Check if the DIFS counter is zero
321                             if (DIFS_counter <= 0) {
322                                 // Indicate that DIFS is done
323                                 DIFS_Ind = 1;
324                                 // initialize the DIFS counter (Redundant but to be sure)
325                                 DIFS_counter = DIFS_time;
326                             } // end of if
327                         } // end of enregy meas
328                     } // End of if contending
329 } // 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 >` `TTTDeliverOfFile_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 notifiys 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 notifiys 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
```

```

405     double PL_meas = 0;
406     if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
407         // initialize a vector pointer to measure PL
408         std::vector<double> *PL = new std::vector<double>;
409         // Measure PL from all the interferer sources
410         PL_measure(ACTIVE_Devices, PL, fc, channel_type);
411         for (auto it = PL->begin(); it != PL->end(); ++it) {
412             // Sum all the received power from all the sources where rx power is tx power*PL
413             PL_meas += Pt_W * std::pow(10, *it / 10);
414         }
415     } // end of active device
416     // compute interference + noise
417     Int_power += PL_meas * (system_time - local_time)
418         + Channel_occupy_time * Noise;
419     std::vector<double> d;
420     // calculate the distance between STA and serving AP
421     d.push_back(
422         std::sqrt(
423             std::pow(coordinates[0] - ((AP*) obj)->
coordinates[0],
424                 2)
425             + std::pow(
426                 coordinates[1]
427                 - ((AP*) obj)->coordinates[1],
428                 2)
429             + std::pow(height - ((AP*) obj)->height, 2)));
430     std::vector<double> *PL_sig = new std::vector<double>;
431     PL_measure(d, PL_sig, fc, channel_type);
432     // calculate the received signal power
433     Sig_power = Channel_occupy_time * Pt_W
434         * std::pow(10, PL_sig->front() / 10);
435     // Calculate SINR and convert to dB
436     double SINR_db = 10 * std::log10(Sig_power / Int_power);
437     // Log the SINR for stats
438     ((AP*) obj)->SINR.push_back(SINR_db);
439     // find if the transmission is in collision based on the SINR
440     bool collision = find_prb_collision(SINR_db, ((AP*) obj)->MCS_DL[ID]);
441     // update global variable total_tti marking finish of 1 tti
442     total_tti++;
443     if (collision == 0) {
444         // if packet not in collision
445         // update the success
446         ((AP*) obj)->success[ID] += 1;
447         // Decrement the data received from the DL buffer
448         ((AP*) obj)->Buffer_DL[ID].front() -=
449             ((AP*) obj)->ALLOC_data_DL[ID];
450         // Decrement the data received from the summ BUFF DL buffer
451         ((AP*) obj)->SUM_BUFF_DL[ID] -= ((AP*) obj)->ALLOC_data_DL[ID];
452         // increment the served data STAT
453         ((AP*) obj)->served_data += ((AP*) obj)->ALLOC_data_DL[ID];
454         // decrement the total buff
455         ((AP*) obj)->TOT_BUFF -= ((AP*) obj)->ALLOC_data_DL[ID];
456         // if the file is finished
457         if (((AP*) obj)->Buffer_DL[ID].front() <= 0) {
458             // record the time taken for the file to download
459             int te2 = system_time
460                 - ((AP*) obj)->Time_file_arrived_DL[ID].front();
461             // push the time taken for STATs
462             ((AP*) obj)->TTTDeliverOfFile_DL.push_back(te2);
463             // pop the time the file arrived
464             ((AP*) obj)->Time_file_arrived_DL[ID].pop_front();
465             // see if there is another file and check if that is scheduled too
466             if (((AP*) obj)->SUM_BUFF_DL[ID] > 0) {
467                 // check the top of the buffer
468                 int temp = ((AP*) obj)->Buffer_DL[ID].front();
469                 // pop it out of the buffer
470                 ((AP*) obj)->Buffer_DL[ID].pop_front();
471                 // ADD it to the next packet
472                 ((AP*) obj)->Buffer_DL[ID].front() += temp;
473             } else {
474                 // due 20us granularity some zero padding will happen
475                 // if there is no new packet remove the zeros from counting into served data
476                 ((AP*) obj)->served_data += ((AP*) obj)->SUM_BUFF_DL[ID];
477                 // do the same for tot buff
478                 ((AP*) obj)->TOT_BUFF -= ((AP*) obj)->SUM_BUFF_DL[ID];
479                 // set the sum buff to zero
480                 ((AP*) obj)->SUM_BUFF_DL[ID] = 0;
481                 // remove the finished packet from buffer
482                 ((AP*) obj)->Buffer_DL[ID].pop_front();
483             } // end of else
484             // decrement that file from the BUF_Ind
485             ((AP*) obj)->BUF_Ind -= 1;
486         } // end of buffer_dl
487         // reset the ALLOC data dl
488         ((AP*) obj)->ALLOC_data_DL[ID] = 0;
489         // MCS Adaptation
490         // if total attempts for this STA are 100

```

```

491         if ((AP*) obj)->attempt[ID] == 100) {
492             // check among the 100 attempts >70 are success
493             if ((AP*) obj)->success[ID] > 70) {
494                 // then increase the MCS
495                 ((AP*) obj)->MCS_DL[ID] += 1;
496                 // if the MCS is >= MAX MCS
497                 if ((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
498                     // then set it to MAX MCS
499                     ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
500             } // end of success
501             // check if the success are less than 30%
502             else if ((AP*) obj)->success[ID] < 30) {
503                 // then decrement the MCS
504                 ((AP*) obj)->MCS_DL[ID] -= 1;
505                 // if the MCS is less than the minimum
506                 if ((AP*) obj)->MCS_DL[ID] < 0)
507                     // then set it to minimum
508                     ((AP*) obj)->MCS_DL[ID] = 0;
509             } // end of if else
510             // set the attempts to 0 for this STA
511             ((AP*) obj)->attempt[ID] = 0;
512             // set the success to 0 for this STA
513             ((AP*) obj)->success[ID] = 0;
514         } // end of attempt condition
515         // reset the ALLCO data dl
516         ((AP*) obj)->ALLOC_data_DL[ID] = 0;
517         // update contention window to the minimum
518         ((AP*) obj)->CW = CW_min;
519     } // end of collision condition
520     // if it is collision
521     else {
522         // double the contention window
523         ((AP*) obj)->CW = ((AP*) obj)->CW * 2;
524         // if the current window is maximum or greater
525         if ((AP*) obj)->CW > CW_max)
526             // set it to maximum
527             ((AP*) obj)->CW = CW_max;
528         // MCS Adaptation
529         // if total attempts for this STA are 100
530         if ((AP*) obj)->attempt[ID] == 100) {
531             // check among the 100 attempts >70 are success
532             if ((AP*) obj)->success[ID] > 70) {
533                 // then increase the MCS
534                 ((AP*) obj)->MCS_DL[ID] += 1;
535                 // if the MCS is >= MAX MCS
536                 if ((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
537                     // then set it to MAX MCS
538                     ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
539             } // end of success
540             // check if the success are less than 30%
541             else if ((AP*) obj)->success[ID] < 30) {
542                 // then decrement the MCS
543                 ((AP*) obj)->MCS_DL[ID] -= 1;
544                 // if the MCS is less than the minimum
545                 if ((AP*) obj)->MCS_DL[ID] < 0)
546                     // then set it to minimum
547                     ((AP*) obj)->MCS_DL[ID] = 0;
548             } // end of if else
549             // set the attempts to 0 for this STA
550             ((AP*) obj)->attempt[ID] = 0;
551             // set the success to 0 for this STA
552             ((AP*) obj)->success[ID] = 0;
553         } // end of attempt condition
554     }
555     // reset the ALLCO data dl
556     ((AP*) obj)->ALLOC_data_DL[ID] = 0;
557 }
558 // if EVENT from serving AP and STA is not in receiving mode and data is intended for this STA
559 else if (temp && ((AP*) obj)->CURRENT_STA_DL == ID) {
560     // double the contention window
561     ((AP*) obj)->CW = ((AP*) obj)->CW * 2;
562     // if the current window is maximum or greater
563     if ((AP*) obj)->CW > CW_max)
564         // set it to maximum
565         ((AP*) obj)->CW = CW_max;
566     // MCS Adaptation
567     // if total attempts for this STA are 100
568     if ((AP*) obj)->attempt[ID] == 100) {
569         // check among the 100 attempts >70 are success
570         if ((AP*) obj)->success[ID] > 70) {
571             // then increase the MCS
572             ((AP*) obj)->MCS_DL[ID] += 1;
573             // if the MCS is >= MAX MCS
574             if ((AP*) obj)->MCS_DL[ID] >= MAX_MCS)
575                 // then set it to MAX MCS
576                 ((AP*) obj)->MCS_DL[ID] = MAX_MCS - 1;
577         } // end of success

```

```

578         // check if the success are less than 30%
579     else if ((AP*) obj)->success[ID] < 30) {
580         // then decrement the MCS
581         ((AP*) obj)->MCS_DL[ID] -= 1;
582         // if the MCS is less than the minimum
583         if ((AP*) obj)->MCS_DL[ID] < 0)
584             // then set it to minimum
585             ((AP*) obj)->MCS_DL[ID] = 0;
586     } // end of if else
587     // set the attempts to 0 for this STA
588     ((AP*) obj)->attempt[ID] = 0;
589     // set the success to 0 for this STA
590     ((AP*) obj)->success[ID] = 0;
591 } // end of attempt condition
592 // find the distance from the AP
593 double d = std::sqrt(
594     std::pow(coordinates[0] - ((AP*) obj)->coordinates[0], 2)
595     + std::pow(coordinates[1] - ((AP*) obj)->
coordinates[1],
596                 2) + std::pow(height - ((AP*) obj)->
height, 2));
597 // remove the AP from the interferers list
598 for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
599     ++it) {
600     if (*it == d) {
601         ACTIVE_Devices.erase(it);
602         break;
603     } // end of if loop
604 } // end of for loop
605 } // end of else if
606 else {
607     double d = 0;
608     // if EVENT from STA
609     if (type == "STA")
610         // Measure the distance
611         d = std::sqrt(
612             std::pow(coordinates[0] - ((STA*) obj)->
coordinates[0], 2)
613             + std::pow(
coordinates[1]
614                 - ((STA*) obj)->coordinates[1], 2)
615             + std::pow(height - ((STA*) obj)->height, 2));
616     if (type == "AP") // if EVENT from AP
617         // Measure the distance
618         d = std::sqrt(
619             std::pow(coordinates[0] - ((AP*) obj)->
coordinates[0], 2)
620             + std::pow(
coordinates[1]
621                 - ((AP*) obj)->coordinates[1], 2)
622             + std::pow(height - ((AP*) obj)->height, 2));
623
624     // if STA is in RX mode
625     if (RX_ind == 1) {
626         // Measure the PL
627         double PL_meas = 0;
628         if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
629             // initialize a vector pointer to measure PL
630             std::vector<double> *PL = new std::vector<double>;
631             // Measure PL from all the interferer sources
632             PL_measure(ACTIVE_Devices, PL, fc, channel_type);
633             for (auto it = PL->begin(); it != PL->end(); ++it) {
634                 // Sum all the received power from all the sources where rx power is tx power*PL
635                 PL_meas += Pt_W * std::pow(10, *it / 10);
636             }
637         } // end of active device
638         // accumulate interference power
639         Int_power += PL_meas * (system_time - local_time);
640         // record the interference change time
641         local_time = system_time;
642         // remove the device from the interferers list
643         for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
644             ++it) {
645             if (*it == d) {
646                 ACTIVE_Devices.erase(it);
647                 break;
648             }
649         } // end of for loop
650     } // end of RX=1
651     else {
652         // remove the device from the interferers list
653         for (auto it = ACTIVE_Devices.begin(); it !=
ACTIVE_Devices.end();
654             ++it) {
655             if (*it == d) {

```

```

658             ACTIVE_Devices.erase(it);
659             break;
660         }
661     } // end of for loop
662 } // end of else
663 } // end of else
664 } // 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 notifs 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 Transmitting 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
320         // intended for this STA
321         // Start receiving
322         RX_ind = 1;
323         // stop contending
324         contending = 0;
325         // Mark the time the transmission started
326         local_time = system_time;
327         // init sig power
328         Sig_power = 0;
329         // init interferer power
330         Int_power = 0;
331         // Record the channel occupy time
332         Channel_occupy_time = ((AP*) obj)->Channel_occupy_time;
333     } // end of temp
334     // else add the event sources to interferers list
335     else {
336         double d = 0;
337         // if the event is from other STA
338         if (type == "STA")
339             // find the distance from the interferer
340             d = std::sqrt(
341                 std::pow(coordinates[0] - ((STA*) obj)->
342                 coordinates[0], 2)
343                 + std::pow(
344                     coordinates[1]
345                     - ((STA*) obj)->coordinates[1], 2)
346                 + std::pow(height - ((STA*) obj)->height, 2));
347         if (type == "AP")
348             // find the distance from the interferer
349             d = std::sqrt(
350                 std::pow(coordinates[0] - ((AP*) obj)->
351                 coordinates[0], 2)
352                 + std::pow(
353                     coordinates[1]
354                     - ((AP*) obj)->coordinates[1], 2)
355                 + std::pow(height - ((AP*) obj)->height, 2));

```



```

353         // check if the current STA is in receiving mode
354         if (RX_ind == 1) {
355             // Measure the PL
356             double PL_meas = 0;
357             if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
358                 // initialize a vector pointer to measure PL
359                 std::vector<double> *PL = new std::vector<double>;
360                 // Measure PL from all the interferer sources
361                 PL_measure(ACTIVE_Devices, PL, fc, channel_type);
362                 for (auto it = PL->begin(); it != PL->end(); ++it) {
363                     // Sum all the received power from all the sources where rx power is tx power*PL
364                     PL_meas += Pt_W * std::pow(10, *it / 10);
365                 }
366             } // end of active device
367             // Add the interferece power of previous state
368             Int_power += PL_meas * (system_time - local_time);
369             // record the time the interference changed
370             local_time = system_time;
371             // push the interferer distance to interferers list
372             ACTIVE_Devices.push_back(d);
373         } // end of RX condition
374         else {
375             // Add the new interferer to the interferers list
376             ACTIVE_Devices.push_back(d);
377         } // end of else condition
378     } // end of else not associated
379 } // 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_BUFFER += 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
142                     us
143                     Counter = uni_distribution(generator) * wifi_slot;
144                     // Set DIFS_Ind to zero as this is the start of DCF procedure
145                     DIFS_Ind = 0;
146                     // Set the DIFS Counter to DIFS duration
147                     DIFS_counter = DIFS_time;

```

```

147         // Initialize energy measurement to thermal noise
148         Energy_meas = Noise;
149         // Measure the energy from all interferers
150         if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
151             // initialize a vector pointer to measure PL
152             std::vector<double> *PL = new std::vector<double>;
153             // Measure PL from all the interferer sources
154             PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
155             for (auto it = PL->begin(); it != PL->end(); ++it) {
156                 // Sum all the received power from all the sources where rx power is tx power*PL
157                 Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
158             } // End of for loop
159         } // end of active devices empty condition
160     } // End of buff ind>0 condition
161 } // End of contending tx rx condition
162 else if (contending == 1) {
163     // if contending then do energy measurements
164     // set jump to 0
165     jump = 0;
166     // Initialize energy measurement to thermal noise
167     Energy_meas = Noise;
168     // Measure the energy from all interferers
169     if (!ACTIVE_Devices.empty()) { // compute only if there are interferers
170         // initialize a vector pointer to measure PL
171         std::vector<double> *PL = new std::vector<double>;
172         // Measure PL from all the interferer sources
173         PL_measure(ACTIVE_Devices, PL, fc, channel_type); // PL is in dB
174         for (auto it = PL->begin(); it != PL->end(); ++it) {
175             // Sum all the received power from all the sources where rx power is tx power*PL
176             Energy_meas = Energy_meas + Pt_W * std::pow(10, *it / 10);
177         } // End of for loop
178     } // end of active devices empty condition
179 } // end of else if contending condition
180 else {
181     // might be receiving might be transmitting
182     // set the jump to zero
183     jump = 0;
184 } // end of else
185 // return time jump value
186 return jump;
187 } // 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 transmitting
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.

```

685         {
686     switch (EVENT) {
687
688     case TRANSMITTING:
689         // Call back AP
690         for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
691             ((AP*) (*it))->callback_tx(this, this->TYPE);
692         }
693
694         // call back STA
695         for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
696             ((STA*) (*it))->callback_tx(this, this->TYPE);
697         }
698
699         break;
700
701     case FIN_TRANSMITTING:
702         // AP call back
703         for (auto it = listener_AP.begin(); it != listener_AP.end(); ++it) {
704             ((AP*) (*it))->callback_fintx(this, this->TYPE);
705         }
706
707         // STA call back
708         for (auto it = listener_STA.begin(); it != listener_STA.end(); ++it) {
709             ((STA*) (*it))->callback_fintx(this, this->TYPE);
710         }
711
712         break;
713     default:
714         std::cout << "unknown event occurred \n";
715     } // end of switch
716
717 } // 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.

```

195         {
196     // DO DCF if contending
197     if (contending == 1) {
198         // See if DIFS is done
199         if (DIFS_Ind == 1) {
200             // check if the measured enrgy is less than WIFI sensing threshold
201             if (Energy_meas > WIFI_TH) {
202                 // If energy meas id > sens thresh then bring the counter to multiple of wifi slots
203                 Counter = Counter - (Counter % wifi_slot) + wifi_slot;
204                 // Set DIFS Ind to 0
205                 DIFS_Ind = 0;
206                 // Set the DIFS counter
207                 DIFS_counter = DIFS_time;
208             } // end of if energy condition
209             else
210                 // Decrement the counter by 1 us If energy meas is less than sensing threshold
211                 Counter = Counter - 1;
212             // Check whether the counter is zero
213             if (Counter <= 0) {
214                 // If counter is zero
215                 // set contending to false
216                 contending = false;
217                 // Set DIFS indicator to zero
218                 DIFS_Ind = 0;
219                 // Set DIFS counter

```

```

220         DIFS_counter = DIFS_time;
221         // Set TX indicator to 1
222         TX_ind = 1;
223         // Start Scheduling
224         // Set channel occupy time to zero;
225         Channel_occupy_time = 0;
226         // check if the buffer is non empty (Redundant condition)
227         if (TOT_BUFF > 0) {
228             // increase the attempts counter
229             attempt += 1;
230             // see how much channel time required for transmission
231             long long time = std::ceil(
232                 (double) TOT_BUFF / WIFI_data_20us[MCS_UL]);
233             // check whether the time required is less than the MAX tx opportunity
234             if (20 * time < MAX_TXOP) { // 20us is the minimum granularity
235                 // update the channel occupy time
236                 Channel_occupy_time = 20 * time;
237                 // Update Allocated data
238                 ((AP*) BS_obj)->ALLOC_data_UL[ID] = TOT_BUFF;
239                 // update the pumped data
240                 pumped_data += TOT_BUFF;
241             } // end of if time condition
242             else {
243                 // occupy the channel for max tx op
244                 Channel_occupy_time = MAX_TXOP;
245                 // update Allocated data
246                 ((AP*) BS_obj)->ALLOC_data_UL[ID] =
247                     WIFI_data_20us[MCS_UL] * MAX_TXOP / 20;
248                 // update pumped data
249                 pumped_data += WIFI_data_20us[MCS_UL] * MAX_TXOP / 20;
250             } // end of else
251         } // end of TOT buf condition
252         else {
253             // print error for debugging logical errors
254             std::cout
255                 << "##### Critical Error: contending when no data ##### "
256                 << std::endl;
257         }
258         // Notify the listening group about this transmission
259         notify_listener(TRANSMITTING);
260     } // end of counter zero
261 } // End of DIFS ind
262 else if (Energy_meas < WIFI_TH) { // if measured energy is less than the WIFI sensing threshold
    (lus)
263     // Decrement DIFS counter by lus
264     DIFS_counter = DIFS_counter - 1;
265     // Check if the DIFS counter is zero
266     if (DIFS_counter <= 0) {
267         // Indicate that DIFS is done
268         DIFS_ind = 1;
269         // initialize the DIFS counter (Redundant but to be sure)
270         DIFS_counter = DIFS_time;
271     } // end of DIFS_counter
272 } // end of enregy meas
273 } // End of if contending
274 } // 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.

```

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         // Associated AP object Handle
47         BS_obj = BS_object;
48         // Minimum Contention window size of DCF
49         CW_min = params.CW_min;
50         // Maximum Contention window size of DCF
51         CW_max = params.CW_max;
52         // slot duration of DCF
53         wifi_slot = params.wifi_slot;
54         // DIFS duration
55         DIFS_time = params.DIFS_time;

```

```

56 // Maximum transmit opportunity in micro seconds
57 MAX_TXOP = params.MAX_TXOP;
58 // Uplink load per user in packets per micro second
59 load_peruser = params.load_peruser_ul;
60 // Noise floor of the receiver in Watts measured over 1 subcarrier
61 Noise = params.Noise;
62 // Transmit power of the AP in Watts measured over 1 subcarrier
63 Pt_W = params.Pt_STA;
64 // Carrier frequency used for Pathloss measurements
65 fc = params.fc;
66 // Channel Model
67 channel_type = params.channel_model;
68 // Packet size at the application
69 file_size = params.file_size;
70 // Wifi Energy Detect Sensing threshold
71 WIFI_TH = params.WIFI_TH;
72 // Maximum MCS value
73 MAX_MCS = params.MAX_MCS;
74 // Set the MCS (MAX) for DL transmission for all the STAs
75 MCS_UL = MAX_MCS - 1;
76 // bits transmitted in 20micro seconds for each MCS
77 WIFI_data_20us.insert(WIFI_data_20us.begin(), &params.WIFI_data_20us[0],
78                       &params.WIFI_data_20us[MAX_MCS]);
79 // Exponential random variable initializer
80 std::exponential_distribution<double> exp_distribution(load_peruser);
81 // time after which next file arrives
82 Time_next_file_arrives = (long long) std::round(
83     exp_distribution(generator));
84 // Initialize attempt
85 attempt = 0;
86 // Initialize success
87 success = 0;
88 // Set the current contention window to the minimum
89 CW = CW_min;
90 }

```

### 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/hdr/802.11ac/STA.h)
- [/home/sreekanthepc/git\\_bitbucket\\_repos/hetnetsim\\_cpp/src/802.11ac/STA.cpp](/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](https://en.wikipedia.org/wiki/Monotone_cubic_interpolation)

Definition at line 107 of file common.cpp.

```

108                                     {
109     int i = 0;
110     // compute dx i.e SNR differences
111     std::vector<double> dx;
112     for (auto it = SNR_LUT->begin(); it != SNR_LUT->end() - 1; ++it) // runs from i = 1...n-1
113     {
114         dx.push_back(*(it + 1) - *(it));
115     }
116     // compute dy i.e BLER differences
117     std::vector<double> dy;
118     for (auto it = BLER_LUT->begin(); it != BLER_LUT->end() - 1; ++it) // runs from i = 1...n-1
119     {
120         dy.push_back(*(it + 1) - *(it));
121     }
122     // compute slopes dy/dx
123     std::vector<double> ms;
124     for (auto it = dx.begin(); it != dx.end(); ++it) // runs from i = 1...n-1
125     {
126         ms.push_back(dy[i] / (*it));
127         i++;
128     }
129     std::vector<double> c1s;
130     std::vector<double> c2s;
131     std::vector<double> c3s;
132
133     // compute c1s first order coefficients

```

```

134     cls.push_back(ms.front());
135     for (i = 0; i < ms.size() - 1; i++) {
136         if (ms[i] * ms[i + 1] < 0)
137             cls.push_back(0);
138         else
139             cls.push_back(
140                 3 * (dx[i] + dx[i + 1])
141                 / (((dx[i] + dx[i + 1] + dx[i + 1]) / ms[i])
142                    + ((dx[i] + dx[i + 1] + dx[i]) / ms[i + 1])));
143     }
144     cls.push_back(ms.back());
145
146     // compute c2s and c3s 2nd and third order coeffs
147     for (i = 0; i < cls.size() - 1; i++) {
148         c2s.push_back(
149             (ms[i] - cls[i] - cls[i] - cls[i + 1] + ms[i] + ms[i]) / dx[i]);
150         c3s.push_back((cls[i] + cls[i + 1] - ms[i] - ms[i]) / (dx[i] * dx[i]));
151     }
152     double BLER_est = 0;
153     bool do_interpolation = true;
154
155     // see if we can avoid interpolation
156     for (i = 0; i < SNR_LUT->size(); i++) {
157         if (*(SNR_LUT->data() + i) == SNR_db) {
158             BLER_est = *(BLER_LUT->data() + i);
159             do_interpolation = false;
160             break;
161         }
162     }
163
164     if (do_interpolation) {
165         // see if the input is not in range
166         if (SNR_db < SNR_LUT->front()) {
167             BLER_est = BLER_LUT->front();
168         } else if (SNR_db > SNR_LUT->back()) {
169             BLER_est = BLER_LUT->back();
170         } else {
171             // start interpolation
172             for (i = 0; i < SNR_LUT->size(); i++) {
173                 if (SNR_db < *(SNR_LUT->data() + i)) {
174                     double y_low = *(BLER_LUT->data() + i - 1);
175                     double x_low = *(SNR_LUT->data() + i - 1);
176                     double diff = SNR_db - x_low;
177                     BLER_est = y_low + (cls[i - 1] * diff)
178                         + (c2s[i - 1] * diff * diff)
179                         + (c3s[i - 1] * diff * diff * diff);
180                     break;
181                 } // end of if
182             } // end of for
183         } // end of else
184     } // end of if
185     return BLER_est;
186 } // 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&lt; double &gt; ACTIVE\_devices, std::vector&lt; double &gt; \* 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                 *it = *it < 3 ? 3.1 : *it; // to make sure the distance is more than 3m
31                 // Raise a warning if the cell size is more than 150m
32                 if (*it > 150) {
33                     std::cout
34                         << "_##_#_WARNING_##_#_: d>150 \n ### InH MODEL applicable only for cells of size <
35                         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                 PL_out->push_back(-PL + PL_shd);
75             }
76         }
77     }

```

#### 4.5.2.2 void PL\_measure ( std::vector< double > *ACTIVE\_devices*, std::vector< double > \* *PL*, double *fc*, std::string *channel\_type* )

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

Definition at line 27 of file PL\_measure.cpp.

```

28                                     {
29
30     if (channel_type == "InH") {
31         // Indoor Hotspot Model
32         InH(ACTIVE_devices, PL, fc);
33     }
34     if (channel_type == "RMa") {
35         // Rural Macro
36         RMa(ACTIVE_devices, PL, fc);
37     }
38     if (channel_type == "UMa") {
39         // Urban Macro
40         UMa(ACTIVE_devices, PL, fc);
41     }
42     if (channel_type == "UMi") {
43         // Urban Micro
44         UMi(ACTIVE_devices, PL, fc);
45     }
46 }
```

## 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](https://en.wikipedia.org/wiki/Monotone_cubic_interpolation)

Definition at line 107 of file common.cpp.

```

108                                     {
109     int i = 0;
110     // compute dx i.e SNR differences
111     std::vector<double> dx;
112     for (auto it = SNR_LUT->begin(); it != SNR_LUT->end() - 1; ++it) // runs from i = 1...n-1
113     {
114         dx.push_back(*(it + 1) - *(it));
115     }
116     // compute dy i.e BLER differences
117     std::vector<double> dy;
118     for (auto it = BLER_LUT->begin(); it != BLER_LUT->end() - 1; ++it) // runs from i = 1...n-1
119     {
120         dy.push_back(*(it + 1) - *(it));
121     }
122     // compute slopes dy/dx
123     std::vector<double> ms;
124     for (auto it = dx.begin(); it != dx.end(); ++it) // runs from i = 1...n-1
125     {
126         ms.push_back(dy[i] / (*it));
127         i++;
128     }
129     std::vector<double> c1s;
130     std::vector<double> c2s;
131     std::vector<double> c3s;
132
133     // compute c1s first order coefficients
134     c1s.push_back(ms.front());
135     for (i = 0; i < ms.size() - 1; i++) {
136         if (ms[i] * ms[i + 1] < 0)
137             c1s.push_back(0);
138         else
139             c1s.push_back(
140                 3 * (dx[i] + dx[i + 1])
141                 / (((dx[i] + dx[i + 1] + dx[i + 1]) / ms[i])
142                    + ((dx[i] + dx[i + 1] + dx[i]) / ms[i + 1])));
143     }
144     c1s.push_back(ms.back());
145
146     // compute c2s and c3s 2nd and third order coeffs
147     for (i = 0; i < c1s.size() - 1; i++) {
148         c2s.push_back(
149             (ms[i] - c1s[i] - c1s[i] - c1s[i + 1] + ms[i] + ms[i]) / dx[i]);
150         c3s.push_back((c1s[i] + c1s[i + 1] - ms[i] - ms[i]) / (dx[i] * dx[i]));
151     }
152     double BLER_est = 0;
153     bool do_interpolation = true;
154
155     // see if we can avoid interpolation
156     for (i = 0; i < SNR_LUT->size(); i++) {
157         if (*(SNR_LUT->data() + i) == SNR_db) {
158             BLER_est = *(BLER_LUT->data() + i);
159             do_interpolation = false;
160             break;
161         }
162     }
163
164     if (do_interpolation) {
165         // see if the input is not in range
166         if (SNR_db < SNR_LUT->front()) {
167             BLER_est = BLER_LUT->front();
168         } else if (SNR_db > SNR_LUT->back()) {
169             BLER_est = BLER_LUT->back();
170         } else {

```



```

171         // start interpolation
172         for (i = 0; i < SNR_LUT->size(); i++) {
173             if (SNR_db < *(SNR_LUT->data() + i)) {
174                 double y_low = *(BLER_LUT->data() + i - 1);
175                 double x_low = *(SNR_LUT->data() + i - 1);
176                 double diff = SNR_db - x_low;
177                 BLER_est = y_low + (cls[i - 1] * diff)
178                     + (c2s[i - 1] * diff * diff)
179                     + (c3s[i - 1] * diff * diff * diff);
180                 break;
181             } // end of if
182         } // end of for
183     } // end of else
184 } // end of if
185 return BLER_est;
186 } // 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.

```

27         {
28     for (auto it = ACTIVE_devices.begin(); it != ACTIVE_devices.end(); ++it)
29     {
30         *it = *it < 3 ? 3.1 : *it; // to make sure the distance is more than 3m
31         // Raise a warning if the cell size is more than 150m
32         if (*it > 150) {
33             std::cout
34                 << "_##_#_WARNING_##_#_: d>150 \n ### InH MODEL applicable only for cells of size <
35             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 }

```

## 4.10 /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.

```

28                                     {
29
30     if (channel_type == "InH") {
31         // Indoor Hotspot Model
32         InH(ACTIVE_devices, PL, fc);
33     }
34     if (channel_type == "RMa") {
35         // Rural Macro
36         RMa(ACTIVE_devices, PL, fc);
37     }
38     if (channel_type == "UMa") {
39         // Urban Macro
40         UMa(ACTIVE_devices, PL, fc);
41     }
42     if (channel_type == "UMi") {
43         // Urban Micro
44         UMi(ACTIVE_devices, PL, fc);
45     }
46 }
```

**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](#).