RZ 3594 (# 99604) 06/06/05
Computer Science 61 pages

Research Report

A Cross-Layer System Simulator for UWB-based Wireless Sensor Network

Božidar Radunović*

IBM Research GmbH
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

*This work was performed while the author was at the IBM Zurich Research Laboratory. For more information on the project, contact Hong Linh Truong (hlt@zurich.ibm.com)

Research
Almaden · Austin · Beijing · Delhi · Haifa · T.J. Watson · Tokyo · Zurich

A Cross-Layer System Simulator for UWB-based Wireless Sensor Network

Božidar Radunović

Februar 27, 2005

Acknowledgements

This work would not be possible without numerous fruitful discussions with my colleagues Hong Linh Truong and Martin Weisenhorn.

Contents

Chapter 1

Introduction

1.1 Motivation

Wireless Sensor Network Simulator (WSNS) is a discrete-event simulator of a sensor network, design for WP 3 of PULSER project. Its goal is to simulate low-data rate, transmit-only, PPM-UWB sensor networks. For more details on network architecture see [5]. In this document we describe the network architecture as implemented in the simulator.

The goal of the simulator is to evaluate performance of different variants of network architecture. It has thus to provide an accurate model of both physical and network layer of a networks while maintaining reasonable simulation times. In addition, the analyzed network scenario requires transmit-only sensors and hence half-duplex communication. This in turn implies an extremly simple network layer. In order to exploit this simplicity, we have decided to write a new simulator, and not to use some of the existing network simulators.

The simulator is a set of C++ classes that implement different parts of a sensor networks: sensors, clusterheads, servers, packets, etc. The simulator takes input configuration from a configuration file and generates a corresponding network. It then performs a discrete event simulation where each event represent a change in a system (packet generation, packet arrival, traffic change, etc.). All the events are logged, and the log file is used latter to produce different statistics and evaluate performance.

In this document we describe the basic principles of a transmit-only UWB wireless sensor network, give an overview of the simulator, give details about the classes, define configuration files and log output, and give several examples on how to write scripts that can explot these information.

1.2 Transmit-only UWB Sensor Networks

There is a recent increase in demand for wireless sensor networks, due to its simplicity, low cost and easy deployment. Those networks can serve for different purposes, from measurement and detection, to automation and process control.

A typical wireless sensor network consists of sensor nodes (SNs) and sinks. SNs are wireless nodes equipped with sensing devices whose goal is to gather data in the environment and transmit it to a central server. It is important that the transmission is wireless, since the number of sensors is typically very large, and cost of deployment of a wired infrastructure is prohibitively expensive.

In order to have a long life time, SNs typically use small transmission powers. The area covered by a sensor network may be large, hence we need intermediate devices to relay data. These devices are called cluster heads (CHs). A CH is a device whose task is to capture transmissions of SNs in its environment, optionally do some limiting processing of the data, and forward it to a central server. One CH is responsible for coordinating a number of SNs: for a network of 100 SNs, we envisage to have less than 10 CHs, depending on the network area. Since there are much fewer CHs than SNs, they can be more expensive. They can rely on more sophisticated wireless technology to transmit data to the central server, or in some cases they can also be wired. In this work we focus on the communication between the SNs and CHs, and we assume all CHs have reliable (wired) links to the central server.

1.2.1 Ultra-Wide Band Physical Layer and Coding

One of the promising physical layer technologies for future wireless networks is the ultra-wide band physical layer. The characteristic of UWB is that it uses a large bandwidth, typically of order of several GHz, which allows it to transfer data at high data rates using low powers. Even though sensor applications typically do not require very high data rates, the whole network may require a high data rate due to a large number of simultaneously transmitting SNs.

One particular implementation of UWB is a pulsed-based UWB physical layer. It consists of sending a very short pulses (of order of 1ns). Additional benefit of this technology, which is derived from radar systems, is an accurate distance estimation. Sensor networks based on pulse-based UWB are location aware, which is an important feature for applications like location tracking and intrusion detection.

Another benefit of pulse-based UWB architecture is a simple transmitter architecture. A typical modulation scheme for such physical layer is 2-PPM. A transmitter needs a pulse generation circuit, and the position of a pulse is a simple function of a transmitted symbol. On the contrary, an alternative UWB technology based on OFDM requires a much more complex transmitter that will generate multiple carrier frequency and distribute the load accordingly.

A pulse-based UWB receiver is a significantly more complex circuit. There are two main types of receivers: coherent and non-coherent. A coherent receiver achieves high data rates, but it needs to estimate the channel impulse response and a very accurate synchronization. On the contrary, non-coherent receiver does not estimate channel and needs less accurate synchronization. It has a simpler architecture, but it yields lower data rates. We assume that SNs and CHs are equipped with a non-coherent UWB physical layer that is described in [8].

Fundamental design parameters of a physical layer are transmitting power, coding and rate. In order to achive long range communication, one has to use high transmission power or powerful codes

to cope with signal attenuation. However, due to regulatory limit, high transmission power implies longer delays between pulses and thus a lower data rate. The same holds for coding: more powerful codes are more error-prone but decrease the rate of communication. Our choice of these parameters are explained in detail in Section 1.3.

1.2.2 Transmit-Only Sensor Nodes

A sensor network comprises a large number of SNs. It is thus important that these nodes are as simple and as cheap as possible. We want a sensor network to have a relatively high data rates and location capabilities, and focus our SNs with pulse-based UWB physical layer. As discussed in Section 1.2.1, pulse-based UWB transmitters are cheap and simple to implement. Nevertheless, even a simpler, non-coherent receiver, requires complex elements, such as synchronization circuit, and may be prohibitively expensive for low-end SNs.

We assume a network of transmit-only SNs, equipped with sensing and transmitting devices. These SNs measure some data and transmit it to CHs. They can use an arbitrary medium access scheme, but this scheme depends only on the available data. The SNs cannot sense the medium nor can they receive any feedback from CHs or other SNs, hence SNs are completely unaware of the global state of the network. This choice of sensor architecture implies that most of the design complexity is in the backend.

1.2.3 System Requirements

Sensor networks are usually low data rate networks, as described in [7]. The main reason is that low traffic, hence low *average* data rates imply low power dissipation and long network lifetime. However, we emphasize that we are talking about low average data rate. The peak traffic may still be high, but only during very infrequent time intervals. A typical sensor traffic thus may vary from a few packets per hour up to 400 kbps for video transmissions. Note that these numbers represent average data rates: sensors will transmit packets at physical layer data rate (which is of order of MBps), and the average rate will depend on time gaps between packets.

A typical network consists of up to hundreds of sensors. Therefore, even if a video transmission from a single sensor is considered low traffic, a simultaneous video transmissions of tens of sensors is several times larger than the rate of the physical layer itself. A network should thus be designed in such a way that it can maximize its performance both during low traffic intervals and high traffic bursts.

We assume there exist low priority and high priority nodes. High priority nodes are located near clusterheads and are expected with high probability to successfully transmit packets. Low priority nodes are expected to deliver packets only when the total traffic is low and may be placed far from cluster heads. This facilitates a deployment of a network and makes it cheaper.

1.2.4 Application Scenarios

In order to better explain system requirements, we illustrate them on on an example of a surveillance system, based on scenario 21 from [7]. An underground car park is filled with sensors. There are several types of sensors. Some sensors are low priority sensors, like temperature and humidity measurements. They have very low traffic (< 10 kbps) and one or a few transmissions can be lost.

Other sensors are high-priority sensors, like seismic, infrared and microphone sensors are used to detect movements of an intruder ($\approx$ 10kbps traffic), and cameras that are transmitting live videos from the area ($\approx$ 400kbps traffic). Typical network of this type consists of 10-100 SNs, and when cameras are active the aggregate rate may go up to several tens of Mbps. The scenario is depicted in Figure 1.1.

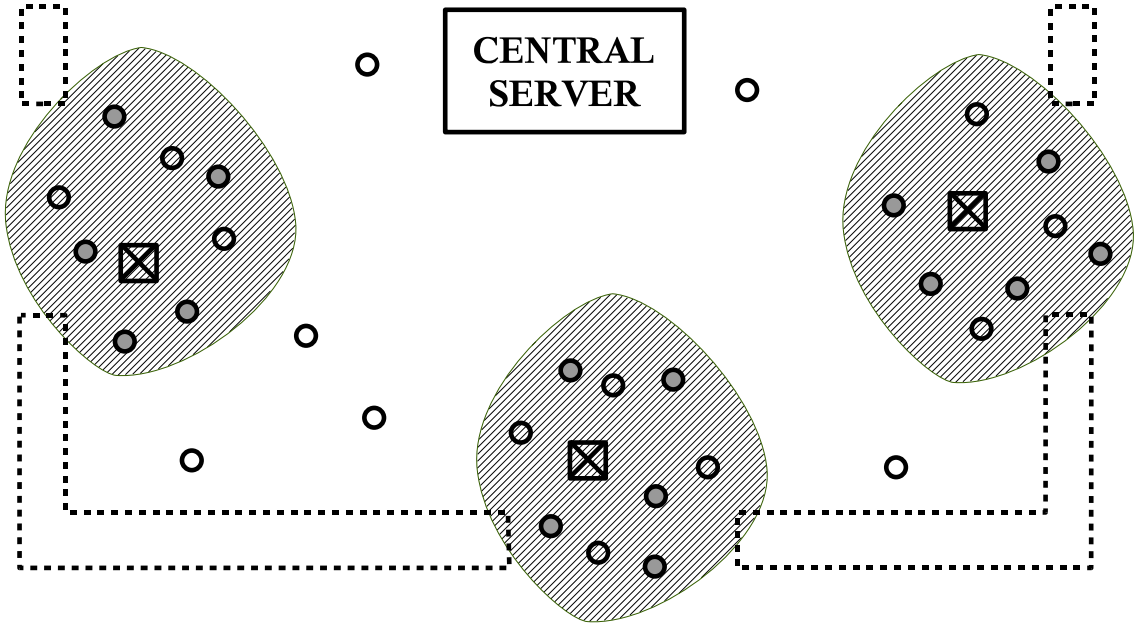

Figure 1.1: An illustration of a transmit-only sensor network in the intrusion detection scenario. SNs are denoted with circles, and CHs with crossed boxes. Empty circles are low-traffic sensors and solid circles are video sensors. Sensors that are placed in shaded areas around cluster heads are high-priority sensors. Others are low-priority sensors.

Camera and movement detection nodes are placed near clusterheads. When a traffic is high, there will be lot of collisions. Sensors are unaware of network traffic, so collisions cannot be prevented. However, if high priority nodes are close to clusterheads, interference from distant transmissions is going to be low compared to received signal power, hence the packet error rates are going to be low. On the contrary, low priority sensors may be significantly farther away. Their packets will be correctly received only when there is no intrusion detection, which is sufficient for this type of application.

Similar frameworks are described in scenarios 15 and 26 of [7]. Scenario 15 discussed position monitoring for training purposes. A typical network contains 100 nodes and maximal rate is 100 kbps. Scenario 26 presents a smart shelf management and monitoring system. The system is required to accommodate up to 1000 nodes with rates of 10-100 kbps. Although hardly ever all nodes will be active at the same time, the total aggregated input traffic of a network can easily go to tens of Mbps.

A similar example is a fire detection sensor network [6]. Sensors are distributed on an area, and their goal is to detect a fire, and to monitor its spreading. The average traffic in such a network is very low. However, in case of fire, there is a burst of traffic from those SNs that detect fire. Most of these information are redundant to some extent: it is sufficient to get packets from one sensor to detect fire. In order to get more precise information on fire spreading, we need to capture more packets.

Another important application parameter is the communication range. As described in [7], a range of communication for this type of applications is from 10m to 100m. We select the target communciation range to be 60m. Network coverage can be further improved by deploying more clusterheads.

1.2.5 Performance Metrics

Summarizing the above requirements, we focus on sensor networks with low average data rates and a large number of nodes, but with high peak data rates. Our goal is to develop network architecture that will be available to sustain the bursts periods of peak transmission rates, defined by these examples, and which will in parallel be efficient in low-traffic periods.

When a traffic is low, collision probability is low. The goal of cluster heads is to capture packets from as many sensors as possible, thus to cover the largest possible area. Therefore, in low-traffic regime, our performance metric is **range maximization**.

On the contrary, when a traffic is high, there will be lot of collisions. All sensors are transmit-only, hence they cannot sense the existing traffic and avoid collisions. In this regime, cluster heads should concentrate only on high-priority sensor in its neighbourhood and maximize the total number of packets they capture from these types of sensors. Thus, in high-traffic regime, our performance metric is **throughput maximization**.

The throughput maximization metric does not explicitly consider fairness issues. By maximizing throughput some distant sensors may starve. However, some form fairness is implied by network topology design. As described in application requirements, high priority sensors are expected to be placed near clusterheads. Therefore, all high-priority sensors will get approximately the same traffic, while only low priority sensors will starve during traffic burst (which is one of the design assumptions). We also propose two additional metrics that address more thoroughly the fairness issue. The first one is **proportional fairness** [4]. Each sensor is assigned a log utility which is log of the number of successfully transmitted packets. The goal is to maximize the sum of log utilities of all sensors. This metric is widely used in networking. The second one is α**-coverage time**. It is the time until at least one packet is received from α fraction of deployed sensors. It depicts capability of clusterheads to extract information from the whole network. All of these metrics can be evaluated in the simulator, and an example is given in scrip files in Section A.2.

1.3 Sensor Architecture

There are three parameters that define sensor arhitecture: packet sizes, transmit power, coding and medium access. Medium access is described in details in [5]. Here we describe the choice of transmit power, coding and packet sizes.

1.3.1 Packet Sizes

Sensors typically send small chunks of data. Here we assume packet size is fixed to 100 bytes (800 bits) with preambles. Similar performance results would be obtained with different packet sizes.

1.3.2 Transmit Power

As defined in Section 1.2.4, we require communication range to be 60m. The upper limit on power spectral density of a UWB signal, defined by FCC, is -41.25 dBm/MHz. The goal is to transmit data at a power sufficiently high such that a clusterhead 60 meters away can synchronize to the signal. As described in [8], synchronization is possible if the bit error rate is lower than 10^{-1}.

We choose pulse energy to be 30 pJ, which implies that the time between consecutive pulses is 400 ns. This in turn yields a rate of 2.5 Mbps. The bit error rate at 60m links is around 10^{-1}. Although this is sufficient for synchronization, it is not enough for successful packet reception. Additional forward error correction or signal combining has to be performed to cope with this error rate. This is described in the following subsection and in Section 1.5.

1.3.3 Coding

As we have seen above, the transmit power is tuned to achieve target bit error rate of 10^{-1} at 60m links. In order to receive a packet at that distance, it is necessary to use some form of forward error correction.

We use a simple model of coding. We assume the underlying channel between a sensor and a clusterhead is a binary symmetric channel [2]. This means that every bit at the output will be flipped with some probability p, called error probability. The capacity of this channel is

$$C(p) = 1 - H(p) = 1 + p\log_2(p) + (1-p)\log_2(1-p).$$

In other words, we can construct a code of infinite block length that will be able to achieve capacity $C(p)$. For example, if error probability $p = 0.1$, the achievable capacity is $C(p) = 0.5$ which means we can transmit 0.5 bits per channel use, or in order to convey n bits of information, we need to transmit $2n$ symbols.

We assume that during a design time we can select a maximum sustainable error rate p. We then construct a code to cope with that error rate. The end-to-end data rate will be the physical data rate multiplied by $C(p)$. Again, if we want to cope with 10% error rate, the end-to-end bit rate will be 1.25 Mbps (the physical data rate is still 2.5 Mbps but in order to transmit a packet of 800 bits we need to send 1600 coded symbols).

Since the maximum tolerable error rate for synchronization is 10%, there is no need to consider codes for higher error rates than that. In the performance analysis part we will evaluate performances of different codes in conjunction with different clusterhead and server architectures.

Note that our model of coding is just a simple approximation. A real implementation would apply coding on soft samples, and not on hard ones, as we assume here. This would increase the performance of codes hence possibly change some of our conclusion. An implementation of coding remains as a future work.

1.4 Clusterhead Architectures

Once a packet is transmitted from a sensor, it will be successfully received at a clusterhead if the signal strength is high enough, and if the level of the interference coming from concurrent transmissions is low enough.

The goal of clusterheads is to successfully receive as many packets from sensors as possible. If a sensor network is lightly loaded, the optimal strategy of a CH is trivial: it should try to decode every packet it sense on the medium. Since CHs do not control sensors' medium access, they cannot prevent transmission failures that occur due to collisions.

However, the story is different when a network load is high. A typical wireless receiver has only a single receiving circuit, thus can receive only one packet at a time. While a CH is receiving a packet from a distant SN, another transmission may starts from a near-by SN. This transmission will interfere and may corrupt the received packet. At the same time, the CH will not be able to receive the interfering packet since its receiving circuit was busy when the transmission started. Hence, both packets will be lost.

In this section we propose two clusterhead architectures that overcome this problem: **adaptive threshold architecture** and **switched architecture**.

1.4.1 CH Architecture Based on Detection Threshold

We first consider a CH with a single receiving circuit and a variable detection threshold, which we denote with P_{dt}. If the signal strength of a received packet is lower than P_{dt}, then the packet will be ignored. Otherwise, the CH will try to receive it. We say that a packet is **detected** if the received signal is stronger than P_{dt}. Only then a CH will try to receive it.

For simplicity of presentation, one can assume that the signal attenuation is a time invariant function of distance and that all SNs send with the same power. Than there exists a threshold region of radius R_{dt}, such that if a SN is outside of this region, a CH will not start receiving packets sent by this SN. This is illustrated on Figure 1.2.

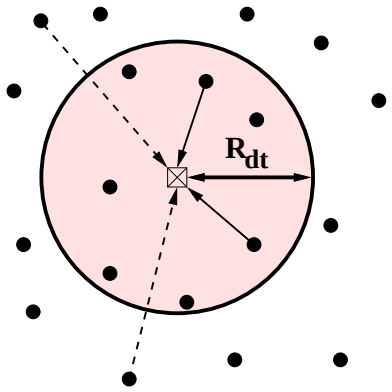

Figure 1.2: An illustration of the detection threshold. Transmissions of sensors whose received signal at the clusterhead is higher than P_{dt} are denoted with solid lines. The transmissions of sensors whose received signal is below P_{dt} are denoted with dashed lines. The equivalent detection region of radius R_{dt} is represented with a shaded circle.

The main reason why a clusterhead does not want to start receiving a packet is that if it starts receiving a packet whose signal strength is low, there is a high probability that the packet will be dropped due to collision. In the meantime, other packets, possibly with a higher signal strength, will be rejected since the only receiving circuit is busy.

An obvious drawback of this approach is that distant SNs will not be able to convey any information at all to the central server. However, this drawback is a consequence of the constraints on the SN architecture, and not of the protocol. Since SNs cannot adapt their medium-access policy, there is no way to receive packets from distant SNs when the traffic is high, regardless of the CH architecture. Nevertheless, this problem can be mitigated in several ways. Firstly, as one can see from the examples in Section 1.2.4, a burst of traffic is usually triggered by an event. Therefore, even if some packets from distant SNs are dropped, we might not loose too much information due to a correlation among data. However, if a reliability of information is crucial, a simple solution is to add more CHs on critical places, to maximize the capture probability. Since the number of CHs is anyway expected to be significantly lower than the number of SNs, the solution will be cheaper than implementing a receiver in each SN.

Theoretical Analysis

In order to better understand this issue, we define a simple model of the system and we analyze it analytically. First we assume that the traffic of every SN is Poisson. This is somewhat reasonable assumption, since if a system is heavily loaded, the optimal medium access for every SN is to defer each transmission for some random time (similar to random backoff in ALOHA). We also assume a simplify physical layer model: if a CH receives a packet from a node at power P^{rcv}, and if an interfering packet, which overlaps even for a small fraction of time, comes with power larger than $P^{rcv} - \Delta$, then the received packet will be lost. We tested this approximation on physical model in [8], using networks with 2 nodes, and we found that approximation holds. This simplification neglects the impacts of multiple concurrent interferers, but as we will see it fits well with the simulated results.

We now consider a scenario depicted on Figure 1.2 with one CH and n SNs. Signal from SN i is received at the CH with power ${P^{rcv}}_i$. Each SN i generate a Poisson traffic with distribution λ_i. Let the detection threshold be P_{dt}, which means that we will try to decode packets whose received power is larger than P_{dt}. The total traffic generated by those nodes is

$$\lambda_a(P_{dt}) = \sum_{i:{P^{rcv}}_i \geq P_{dt}} \lambda_i.$$

We first estimate the probability that the receiver is idle at any given moment in time. The receiver is idle if it has finished decoding a previous packet (successfully or unsuccessfully), and if no other packet has arrived in the meantime with received power larger P_{dt}. The state of receiver (busy or idle) is a stationary process in time so we call the probability of receiver being idle $P_{\text{rcv idle}}$. We can describe this process with a continuous Markov chain and we get the stationary probability $P_{\text{rcv idle}} = 1/(1 + \lambda_a(P_{dt}))$.

We next model the probability that a SN i will successfully transmit a packet. This will happen if the receiver is idle at the time a packet transmission starts, and if the packet does not overlap with any packet whose received power is higher than ${P^{rcv}}_i - \Delta$. We assume all packets have a fixed length and we assume that the packet transmission time is one. Similar to non-slotted Aloha, we have that

if a packet arrives at time 0, then any other packet arriving within $[-1, 1]$ will interfere with it and cause collision. Therefore, the packet capture probability of node i is

$$P_{\text{capture}}(i, P_{dt}) = P_{\text{rcv idle}}(P_{dt}) \exp(-2 \sum_{j:P^{rcv}{}_j \geq P^{rcv}{}_i - \Delta} \lambda_j).$$

The average throughput of SN i is then $\lambda_i P_{\text{capture}}(i, P_{dt})$ and the average throughput of all SNs is

$$\bar{X}(P_{dt}) = \sum_{i:P^{rcv}{}_i \geq P_{dt}} \lambda_i P_{\text{capture}}(i, P_{dt}) \tag{1.1}$$

The optimization problem of maximizing (1.1) can be solved numerically. We solved it for a large number of topologies and traffic distribution and we find that it is always optimal to maintain $\lambda_a = 0.75$ which yields the efficiency of the medium $\bar{X}$ of 22%. In other words, we should estimate $\lambda_a(P_{dt})$ and vary P_{dt} to obtain $\lambda_a = 0.75$.

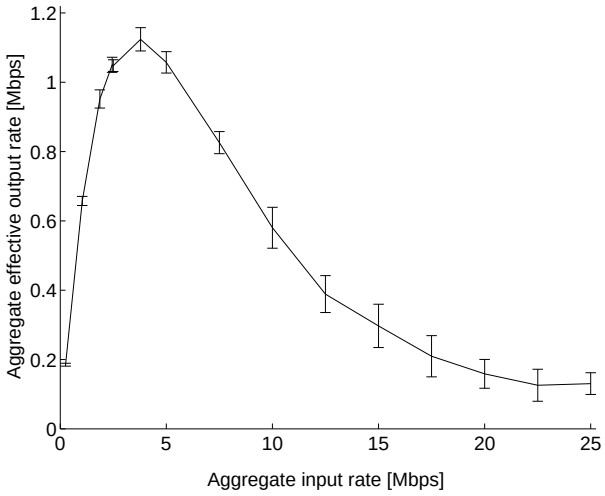

Figure 1.3: We consider 50 SNs uniformly distributed on 40m$\times$40m square, with one CH in the middle. On x axis we see the aggregate input SN traffic. On y axis we see the aggregate goodput.

We verify our model by simulations. We randomly distribute 50 SNs on 40m x 40m square and look at the goodput of the system for different load. The physical link rate is 5Mb/s, and we can see in Figure 1.3 that the goodput is maximal when the aggregate traffic is around 75% of the physical link rate. At that point, the utilization of the system is around 22%.

It may seems at the first sight that a simple model of non-slotted Aloha can be use to model the problem. However, in non-slotted Aloha, the maximum utilization is 18%, which is achieved when the total load is 50% of the physical fixed rate. These numbers have a high discrepancy with the simulation results from Figure 1.3, hence they cannot be used to design an efficient adaptive receiver.

Optimal Architecture of Adaptive Receiver

As described in Section 1.4.1, it is optimal to keep $\lambda_a = 0.75$. We propose a simple method to track load λ_a and utilization $\bar{X}$ of the system, and to adapt P_{dt} in order to keep utilization at the maximum. We give a simple example of packet arrivals in Figure 1.4 to illustrate the idea.

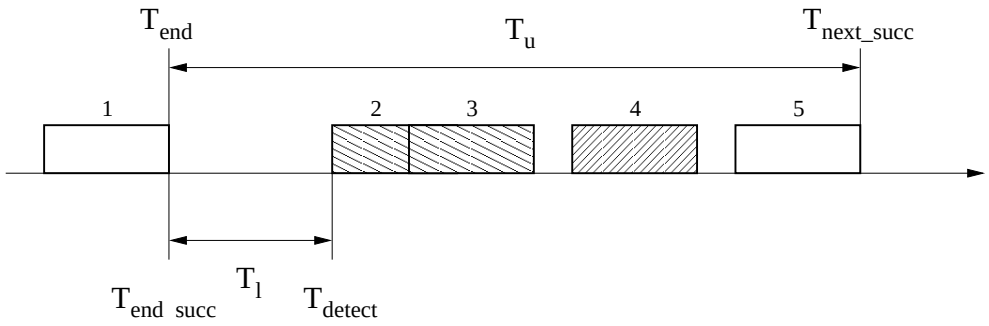

Figure 1.4: An example of packet arrivals that illustrates the adaptation mechanism. Packet 1 has arrived and is well received. Packet 2 was being received when it collided with packet 3 and is discarded. Packet 4 is not detected because it arrives from a node outside of a detection threshold. Finally, packet 5 again is well received.

We first show how to estimate $\lambda_a, \bar{X}$. As shown in Figure 1.4, we denote with T_l the average idle time of a receiver, that is the average time between two packets from sensors that are detected. We keep track of the end time T_{end} of the last detected packet (successfully or unsuccessfully received) and at the moment T_{detect} when we detect a new packet, we update $T_l = \alpha T_l + (1-\alpha)(T_{\text{detect}} - T_{\text{end}})$. The estimate of the intensity of detected load is then $\lambda_a = \text{packet\_duration}/T_l$.

Similar thing is done for estimating utilization $\bar{X}$. We denote with T_u the average time between two successful receptions. We then keep the time of the end of the last successful packet transmission $T_{\text{end\_succ}}$. At the instant of the next successful packet transmission $T_{\text{next\_succ}}$, we update T_u. For updating, we use exponential weighted average $T_u = \alpha T_u + (1-\alpha)(T_{\text{next\_succ}} - T_{\text{end\_succ}})$. The estimate of the utilization is then $\bar{X} = \text{packet\_duration}/T_u$. We set the filter constant $\alpha = 0.95$ in both cases.

In parallel, a CH also needs to learn about existing SNs and their distances. This is done during packet receptions. Note that a CH does not need necessarily to successfully receive a whole packet to perform this estimate. It might be sufficient to decode the header, and to estimate signal strength. As a result of this estimation a CH keeps a list of active sensors and their received powers $\{i, {P^{rcv}}_i\}_{i=1,\cdots,n}$, ordered decreasingly by powers.

The key idea of the algorithm is to keep utilization at 22% and detected load at 75%. In theory it should be sufficient to use detected load as an estimator, but we use both detected load and utilization to have better robustness. First we set $P_{dt} = 0$ and we are able to detect any SN. We start receiving packets, and we update T_l, T_u and the list of SNs. Initially, at the bootstrap, the estimated detected load and utilization are low. Once the detected load goes over 75% and at the same time the utilization drops below 22%, it means that we have passed over the top of the curve on Figure 1.4, hence P_{dt} is too small and has to be increased.

The decision on the size of the detection threshold happens every time a new packet is sensed on the medium. It is important to notice that it is more dangerous to overestimate the detection threshold than to underestimate it. This is due to the shape of the curve on Figure 1.3. If we overestimate the detection threshold when the total input traffic is low (left side of the curve), this means that we further decrease the input traffic hence further decrease the effective output. Similarly, when the total input traffic is high (right side of the curve), if we underestimate the detection threshold, we increase the total number of detected packets and again decrease the effective output rate. However, as we can

see from Figure 1.3, the slope of the curve is much steeper for lower input rates, hence the potential loss when overestimating the detection threshold is higher.

Therefore, we perform a conservative decrease of P_{dt}: if it happens 4 times in a row that the detected load is higher then 75% and the utilization lower than 22%, only then we will increase the detection threshold by removing one SN from it (in other words, if we assume that $P_{dt} = {P^{rcv}}_i$, then we update $P_{dt} = {P^{rcv}}_{i-1}$, if $i > 1$).

On the contrary, when decreasing the detection threshold, we are less conservative. The first moment when the detected load is lower then 75% and the utilization is lower than 22%, we set $P_{dt} = {P^{rcv}}_{i+1}$ (if $i < n$, or else $P_{dt} = 0$).

Another important point is to keep updating T_l and T_u even when no packet arrivals are being detected. In case when a traffic intensity suddenly drops, or nearby nodes cease transmitting, we might have the detection threshold too high. If a new packet arrives at time T_{now}, we will take the following values of $\lambda_a, \bar{X}$:

$$\begin{aligned} \lambda_a &= \frac{\text{packet\_duration}}{\alpha T_l + (1-\alpha)(T_{\text{now}} - T_{\text{end}})}, \\ \bar{X} &= \frac{\text{packet\_duration}}{\alpha T_u + (1-\alpha)(T_{\text{now}} - T_{\text{end\_succ}})}. \end{aligned}$$

This way, the detection threshold will gradually drop in time while there is no detected packet.

At the end, for completeness, we give the pseudo code of operations. Operation `start_transmission(j)` is called at a clusterhead when a packet from node j is sensed. Operation `end_transmission(j)` is called when a packet transmission is finished. Note that `end_transmission(j)` is called only if a packet was detected (the received power is above the detection threshold P_{dt}).

start_transmission (from node j):

```
lambda_a = packet_duration
         / (ALPHA * T_l + (1-ALPHA) * (now - T_end));
X = packet_duration
  / (ALPHA * T_u + (1-ALPHA) * (now - T_end_succ));


if (X < 0.2) count = count + 1;
else count = 0;

if (lambda_a < 75% and X < 20%)
begin
  i = i+1;
  P_dt = Prcv_i;
  count = 0;
end
else if (count >= 4 and lambda_a > 75% and X > 20%)
begin
```

```
    i = i-1;
    P_dt = Prcv_i;
    count = 0;
  end

  if (Prcv_j >= P_dt)
  begin

    // Receive

    T_l = ALPHA * T_l + (1-ALPHA) * (now - T_end);
    lambda_a = packet_duration / T_l;

  end
```

end_transmission:

```
  if (successfully_received)
  begin
      t_u = ALPHA * t_u + (1-ALPHA) * (now - last_rcv);
      util = packet_duration / utime;

      T_end_succ = now;
  end

  if (packet_was_detected) T_end = now;
```

1.4.2 Switched CH Architecture

As we have seen in the previous section, the main reason for low efficiency of a CH is that if it starts receiving a weak packet, and a stronger packet arrives during this reception, the weaker packet will be dropped due to the interference, and the stronger packet will not be received because the receiving circuit was busy when the packet arrived.

The most general way to solve this problem is to include several receiving circuits in parallel so that a CH can cope with all arriving packets. This is not necessary in most of the cases. We propose an alternative solution that offers a similar performance while being simpler and cheaper to implement.

The basic idea is to include another circuit for detection and synchronization, in addition to the full receiving circuit. This additional circuit is constantly monitoring the wireless medium for a newly arriving packets. If a transmission of new packet starts, if its signal is stronger than the signal of the packet currently being received, and if it significantly overlaps with the current packet, the CH stops receiving the current packet and switches to the new, stronger packet. We call this architecture *switched CH architecture.*

1.5 Server Architecture

The goal of a central server is to collect information from clusterheads about received packets. In its most simple implementation, a server only receives packets that are successfully decoded by at least one clusterhead. If no clusterhead successfully decoded a packet, the packet is lost.

In order to improve the performance of the system, we also propose a more advanced server architecture. It is based on ideas from multi-antenna systems. Several clusterheads connected to a central server can be viewed as a multiple input antenna system. If a clusterhead cannot decode the apcket, it just send demodulated soft samples to the central server.

If at least one clusterhead successfully decodes the packet, there is no need for further processing. However, if all of them fail, the server then combines the received samples from multiple clusterheads and tries decoding it.

It is known that the optimal combining for channels with additive white Gaussian noise is *maximum ratio combining* [1]. As we can see from the analysis in [8], our physical layer can be closely approximated with a 2-PAM channel with Gaussian noise, hence maximum ratio combining should also be the optimal combining.

In presence of interference, the interference is no more Gaussian, hence the maximum ratio combining is not anymore the optimal combining. More advanced techniques, like minimum mean-square error (MMSE) receivers should be applied.

Nevertheless, this approach is difficult to pursue for two reasons. Firstly, it is difficult to derive the optimal receiver in case of interference, since the interference introduces the mixed terms (as explained in [8]), and is not purely Gaussian. Secondly, to design an optimal receiver we would need the perfect estimate of total interference at any point in time, which is difficult to implement.

Finally, as we will see in the performance evaluation section, combining is used to increase the range in case of low traffic. In that case, the dominant noise component is background Gaussian noise. When the traffic is high, clusterheads will focus on nearby sensors, hence there will be no use in combining. For the above reasons we do not analyze more advance combining schemes but we focus solely on maximum ratio combining.

Chapter 2

Simulator Description

2.1 Overview

In this chapter we turn to the simulator. We first give an overview of the simulator architecture, and explain the most important parts like event processing, physical layer and coding. Later we give detail descriptions of class hierarchy.

The network consists of two types of nodes. One type of nodes are sensors and the other type are clusterhead. Sensors are transmit-only devices and they periodically send packets. The goal of clusterheads is to receive sensor packets. They do not transmit in wireless medium. Instead, they use wired connections to forward data to the central server.

We first give a brief overview of simulator architectures. It comprises the following classes:

- **Sensors** - defines sensor nodes, that transmit packets to the wireless medium.
- **Clusterheads** - defines clusterhead nodes, that receive packets from the wireless medium. Each clusterhead can hear all the packets.
- **Server** - defines the central server that collects all the data. Each clusterhead passes data to the central server through wirelines (which is implemented such that each clusterhead calls a corresponding method of Server class).
- **Traffic Generator** - generates packet for a sensor according to a predefined distribution. Each time a generator generates a packet, it puts it in a sensor's queue and sets the next time it will generate a packet. Each traffic generator has to be attached to one server at some point in time. In some cases we want to change a traffic of a sensor during a simulation (e.g. before an event occures we have very low data rate traffic, and afterwards we have a very high one). We implement a traffic change in the simulator by attaching a new traffic generator (with different characteristics) at the point in time when the traffic changed.If a new generator is later attached to the same sensor, the old generator is destroyed.
- **Channel** - keeps the parameters of the wireless channel, such as attenuation, rate and bandwidth.
- **Packet** - an object that keeps the information about a transmitted packet, such as packet content and the source id.

- **Event** - each event in the system, such as packet generation, packet transmission, reception, is described with an event object.
- **Scheduler** - keeps a list of all events in the system, sorted according to the event times. Dispatches events one by one in the chronological order.
- **Node** - superclass, from which both sensor and clusterhead classes are inherited.

Network topology is defined by a configuration file, and it is created during the startup. Thus at the very beginning, the simulator creates a channel object, a server object, all nodes (sensors and clusterheads). A scheduler is also at the startup. Traffic generators for each sensor are assigned during lifetime at time instants defined in the script file. If a traffic pattern should change at some point in time, it is sufficient to assign a new traffic generator for that node, which will automatically replace the old one. Packet objects are created by traffic generators and destroyed by central server (whether or not they are correctly received). A life cycle of a packet is depicted on Figure 2.1. Dynamics is depicted on Figure 2.2. A detailed description of a configuration file is given in Section 4.2.

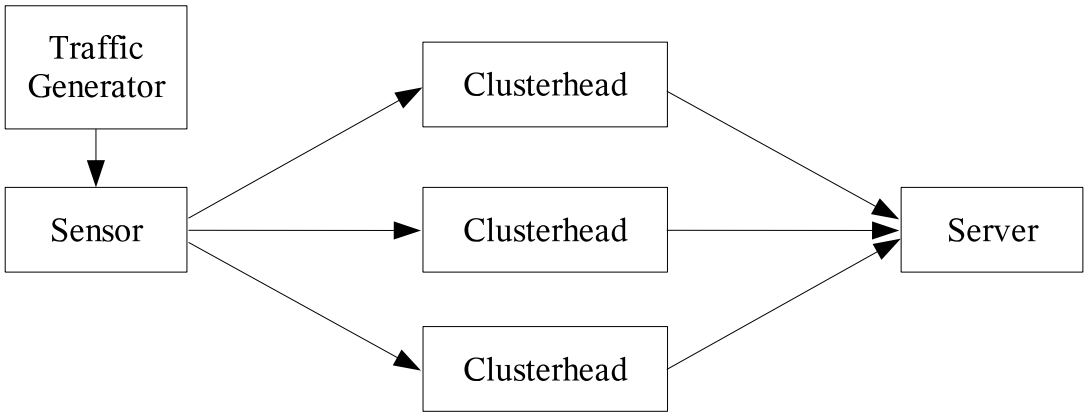

Figure 2.1: Packet life cycle: generated by the traffic generator, it is forwarded to the sensor's queue. It is then broadcasted and observed by all clusterheads. Decoding information is forwarded to the central server which decides whether the decoding is successful.

2.2 Scheduling and Events

The main part of the simulator is the scheduler. It handles all the events in the system. While parsing a configuration file, the simulator inserts the initial events in the scheduler's queue. The scheduler starts processing these events, which in turn generate further events, until an event that stops the simulation occurs.

There are five types of events in the simulator. Event **PKT_CREATE** is scheduled when a packet is to be created (and it creates an actual packet), event **PKT_START** is scheduled when a packet transmission is to start, **PKT_END** is scheduled when a packet transmission is about to finish, event **TRAFFIC_CHANGE** is scheduled when there is a change in traffic pattern and event **SIMULATION_END** is scheduled when the simulation should be ended. The action to be performed when

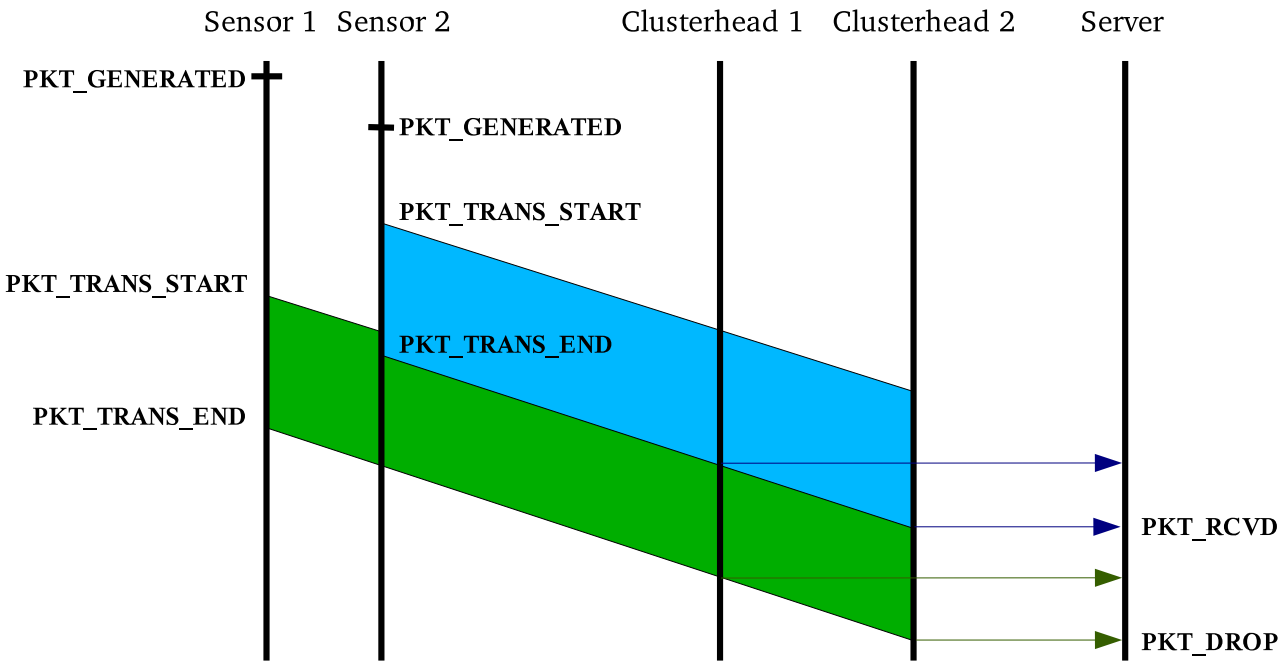

Figure 2.2: Both sensors 1 and 2 transmit a packet. Sensor 2 is closer to clusterheads hence they will detect its packet first. The packet from sensor 1 interferes, but does not destroys the reception, hence packet 2 is received. Packet 1 is lost since no one was listening to it.

an event is executed depends on the event type. Type of event is given in variable **type** of each event object. Next we give a detail list of event types (defined in *scheduler.h*, in type **EventType**).

- **PKT_CREATE** - Packet is being created.

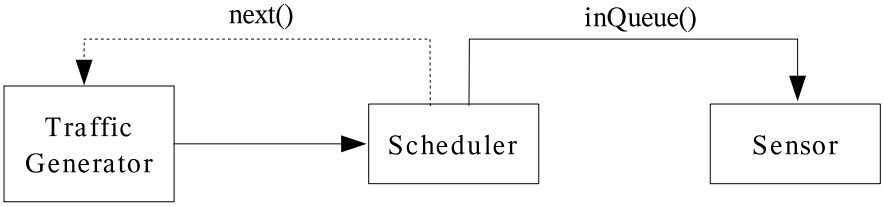

Figure 2.3: Schematic illustration of **PKT_CREATE** event.

Parameters:

- **packet**: a new packet with the payload, generated by traffic generator, which is to be inserted in the sensor's queue.
- **traffic generator**: the traffic generator that has created the packet. This field is used to verify if the generator has changed in the meantime (see Scheduler description for more info).
- **time**: time at which the packet is inserted in the sensor's queue.

Traffic generator decides when a new packet is generated. It first creates **packet**, and then decides at what **time** instant the new packet will be placed in the sensor queue. Then, it creates an event of type **PKT_CREATE** with parameters **packet** and **time**, and inserts it in the event list.

When this event is processed by the scheduler, scheduler calls the *inQueue* method of the corresponding sensor to store the packet in its queue. It then calls the method *next* of the traffic generator. This method will then create a new **PKT_CREATE** event that will correspond to the next packet to be generated.

Note: an event of type **PKT_CREATE** is associated to a particular traffic generator object. Before generating a new event of this type, the scheduler verifies if the corresponding traffic generator is still active on its sensor. If this is not the case, than the new **PKT_CREATE** event is not created, since there is another traffic generator assigned to the sensor. This can happen when a change of a traffic generator (e.g. from constant to exponential) is requested in the configuration file. Once a new generator is active, the old one should not generate packets anymore.

Event processing code:

```
e->getPacket()->getSrc()->inQueue(e->getTime(), e->getPacket());
if (e->getPacket()->getSrc()->getTraffic() == e->getTraffic())
    e->getPacket()->getSrc()->getTraffic()->next(e->getTime());
break;
```

- **PKT_START** - A sensor starts transmitting a packet. There are two subtypes of this event (see below).

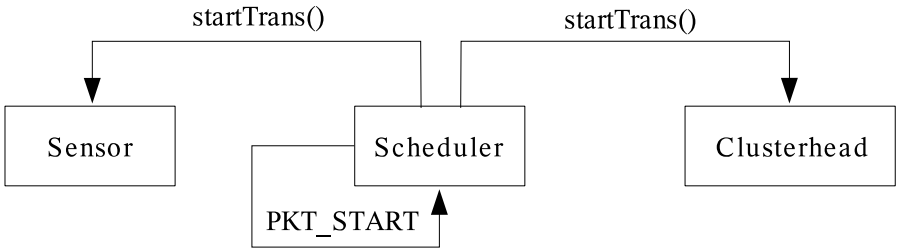

Figure 2.4: Schematic illustration of **PKT_START** event.

Parameters:

  - **packet** - packet a sensor starts transmitting.
  - **time** - time at which the event occurs (packet transmission is started).
  - **clusterhead** - if this field is not empty, that means that **clusterhead** is starting to hear the packet. This event is delayed w.r.t. the start of packet transmission at the sensor since there is a signal propagation delay.

There are two subtypes of this event. If parameter **clusterhead** is NULL, then this event means that at time **time**, a sensor is starting to transmit **packet**. The second subtype is when **clusterhead** is not NULL. This means that **clusterhead** is detecting the start of transmission of packet **packet**. This event is delayed w.r.t. the start of packet transmission at the sensor due to signal propagation delay.

When an event of type **PKT_START**, for a given packet, is processed by the scheduler for the first time (which implies parameter **clusterhead** is set to NULL), it first calls *startTrans* method of the parent sensor. This method puts the corresponding sensor in the busy state, and creates the **PKT_END** event corresponding to the time when the packet transmission will be over. The time of this event depends on the packet size, and is decided by the sensor.

After method *startTrans* is invoked, the scheduler creates several **PKT_START** events for the same packet, one instance for each clusterhead registered in the system. Each event will occur with a delay that is equal to the time the signal travels from the sensor to the corresponding clusterhead.

Finally, if an event of type **PKT_START** that occurs has a non-null **clusterhead**, then the event signifies that **clusterhead** has started receiving the signal. The scheduler than calls method *startTrans* of the clusterhead. The goal of *startTrans* is to maintain a list of the active packets, including the packet being received and those interfering with it. For more details see documentation for class Clusterhead.

Event processing code:

```
if ((ch = e->getClusterhead()) == NULL){
  e->getPacket()->getSrc()->startTrans(e->getTime());
  for(iter = simulator->chs.begin(); iter != simulator->chs.end(); ++iter)
    schedule(new Event(PKT_START, e->getPacket(), (*iter)),
             e->getTime() + (*iter)->delay(e->getPacket()->getSrc()));
} else {
  ch->startTrans(e->getTime(), e->getPacket());
}
```

- **PKT_END** - A sensor ends transmitting a packet. There are two subtypes of this event (see below).

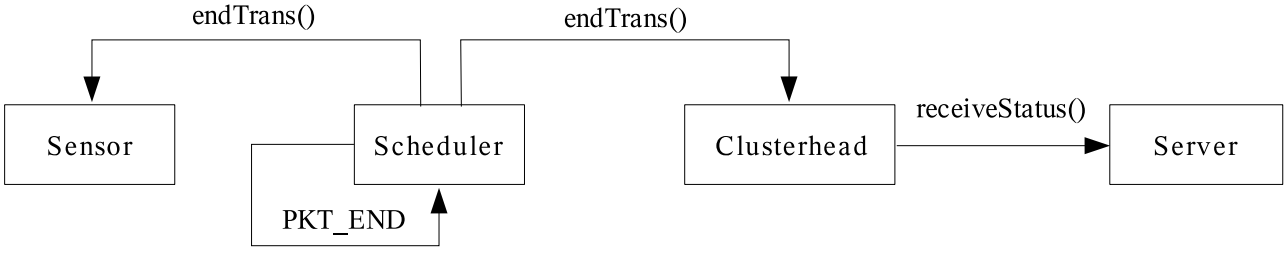

Figure 2.5: Schematic illustration of **PKT_END** event.

Parameters:

– **packet** - packet a sensor stops transmitting.

– **time** - time at which the event occurs (packet transmission is finished).

– **clusterhead** - if this field is not empty, that means that **clusterhead** detects the end of the transmission of the packet. This event is delayed w.r.t. the end of packet transmission at the sensor since there is a signal propagation delay.

There are two subtypes of this event. If parameter **clusterhead** is NULL, then this event means that at time **time**, a sensor is starting to receive **packet**. The second subtype is when **clusterhead** is not NULL. This means that **clusterhead** is detecting the start of reception of packet **packet**. This event is delayed w.r.t. the start of packet transmission at the sensor due to signal propagation delay.

Event processing code:

```
if ((ch = e->getClusterhead()) == NULL){
  e->getPacket()->getSrc()->endTrans(e->getTime());
  for(iter = simulator->chs.begin(); iter != simulator->chs.end(); ++iter)
     schedule(new Event(PKT_END, e->getPacket(), (*iter)),
              e->getTime() + (*iter)->delay(e->getPacket()->getSrc()));
} else {
  ch->endTrans(e->getTime(), e->getPacket());
}
```

- **TRAFFIC_CHANGE** - This event denotes that a sensor has changed the traffic generator. For example, a user may want to change traffic pattern at time instant 5s from constant traffic to an exponential one. Then, an event of type **TRAFFIC_CHANGE** is inserted in the scheduler at time 5s, which will create a new traffic generator and destroy the old one.

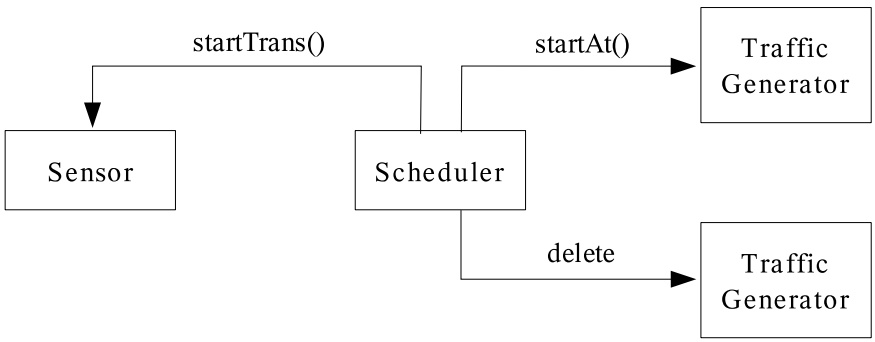

Figure 2.6: Schematic illustration of **TRAFFIC_CHANGE** event.

Parameters:

– **sensor** - sensor that is to change the traffic generator

– **traffic generator** - the new traffic generator to be installed

– **time -** time at which the event occurs (traffic generator is changed).

At **time**, **sensor** changes its traffic generator to **traffic generator**. The new one is destroyed. However, one more packet of the old generator may appear if it has already been put in the queue (through **PKT_CREATE** message), which is usually the case.

Event processing code:

```
if (e->getSensor()->getTraffic()) delete e->getSensor()->getTraffic();
e->getSensor()->setTraffic(e->getTraffic());
e->getTraffic()->startAt(e->getTime());
```

- **SIMULATION_END -** When this event occurs, the scheduler stops the simulator.

  Parameters:

  – **time -** time at which the simulator should be stopped.

2.3 Physical Layer and Coding

2.3.1 Physical Layer

The goal of this simulator is to provide an accurate simulation of a physical layer. Therefore, the networking part is kept simple so that we can increase complexity in physical layer part while still having reasonable simulation time.

It is possible to simulate the actual payload of a packet. It is sufficient to set *has_data* variable of class **Traffic** to 1, which will set a variable of the same name of class **Packet** to 1, and the constructor of **Packet** will create a payload. This payload is currently random and this is sufficient for most of the simulations.

When a clusterhead receives a packet, it adds noise and interferences from overlapping packets to the received signal. At any point in time the simulator is aware what packets are active on the medium, and what bits overlap, so it is able to do an accurate simulation of a physical model by providing **Clusterhead** class with corresponding information. A clusterhead then makes a decision on the signal level as it is done in a real receiver circuit. Thus, the decision might alter a symbol from a packet. Once all symbols are demodulated, we count the number of alerted symbols and calculate the symbol error rate. If this rate is higher than the maximum allowed, the packet is declared corrupted and dropped. One physical layer implemented in the simulator is non-coherent UWB PPM physical layer, described in details in [8].

2.3.2 Coding

The simulator currently implement a basic support for coding. After demodulating a symbol, physical layer returns a hard bit decision (0 or 1) on the value of a symbol. We count a number of alerted bits and if the symbol error rate (SER) is higher than the maximal one *max_error*, the packet is corrupter.

The maximum is defined by user. However, a higher tolerable SER usually implies lower data rate. Since we do not implement a real code, we use a model of a binary symmetric channel (BSC) to [2]. This channel assumes there is a random bit flipping with probability p. Then the capacity of the channel is

$$C(p) = 1 - H(p) = 1 + p\log_2(p) + (1-p)\log_2(1-p).$$

Suppose that the maximum error rate *max_error = 0.1*. This mean we want to produce a code for BSC that will cope with $p = 0.1$ flipping rate. The rate of that code (which is equal to the capacity of the BSC channel) is $C(0.1) \approx 0.5$. In other words, if we have a packet of 1000 bits, we need to code it in a packet of $1000/C(p) = 2000$bits in order to tolerate symbol error of 10%. This is clearly a suboptimal approach as a more sophisticated coding could be implemented on soft output of demodulator. More advanced coding remains as a future work.

The implementation then goes as follows: traffic generator is initialized with uncoded packet size and desirable maximum symbol error rate. Using BSC capacity, traffic generator calculates coded packet size and creates packets of that size. The actual payload in the packets is random, and is used only to test SER. At the decoding end, clusterhead counts number of alerted symbols and compare it with *max_error*, discarding faulty packet.

Chapter 3

Class Structure

In this section we give a list of all classes with belonging members. Each class and each member of each class is documented. Some members are skipped, in case they are simple enough so there is no need to document them.

3.1 Event

Class **Event** defines event objects that are being scheduled. Each event consists of the arrival time and the action to be performed. These objects are put in the scheduler, and executed in the order of the arrival time. When an event is executed, it means that the system time equals the object arrival time.

Class **Event** contains the following members:

- `static long free_id` - next free event ID that should be allocated to the new event.
- `long id` - ID of the event
- `Time time` - time at which this event occurs.
- `EventType type` - event type
- `Packet *pkt` - packet associated to the event (NULL if no packet)
- `Clusterhead *clusterhead` - clusterhead associated to the event (NULL if no clusterhead)
- `Sensor *sensor` - sensor associated to the event (NULL if no sensor)
- `Traffic *traffic` - traffic generator associated to the event (NULL if no traffic)

3.2 Scheduler

Class **Scheduler** implements an ordered list of events. It performs basic operations on events, such as inserting an event in the event list (also called *scheduling*), and processing the first event from the

list (also called *handling*). Scheduler cannot exist alone, and is always associated to an object of type **Simulator**.

Scheduler contains the following members:

- `void schedule(Event *event, Time time)` - Inserts a new **event** in the list of events, at **time**. The event will be processed when the current time of the system coincides with the time of the event.
- `void print()` - prints the list of all scheduled events.
- `int handleEvent()` - Processes the first event in the event queue. This method is called when one event is processed and the simulator wants to process the next available event. The scheduler than takes the most recent event from the queue, set the current simulator time to be equal to the time of that event, and processes the event (see the descriptions of the event types for more details on processing different events).
- `Time time` - current time of the system, set to the time of the event currently being executed.
- `SchedList *start, *end` - pointers to the beginning and the end of the event list.
- `Event *nextEvent()` - private method that returns the next event in the list. Used in handleEvent().
- `Simulator *simulator` - pointer to the simulator owing the scheduler.

3.3 Packet

Class **Packet** defines a concept of a packet. It contains data about a packet, including its size, transmitted power, source node, and the actual data if applicable. If a packet with actual data is generated, the data is generated at random in the constructor.

Packet contains the following members:

- `static long last_id` - ID of the last packet. Used by constructors to create new packets.
- `long id` - the ID of the packet.
- `Sensor *src` - pointer to the sensor that is the source of the packet.
- `long int length` - defines the size of the packet in bits. This represents a size of an encoded packet (if coding is used).
- `long double power` - transmitted power of the packet. From a packet perspective, it is only a number. The way it is used is defined in clusterhead, and may vary from one implementation of a clusterhead to the other.
- `int has_data` - 1 if packet contains data. Data bits are stored in ***data**. Otherwise, **has_data** is 0, and ***data** = NULL;

- `char *data` - pointer to the actual payload.
- `double max_error` - maximum symbol error rate that can occure without corrupting the packet. Currently we implement only a simple model of coding. For more details see Section 2.3.
- `int getDataBit(int i)` - extracts the i-th data bit from the payload.

Packet memory handling:

A packet is created in **Traffic** class, in method *getPacket*. The pointer to this structure is then passed throughout the simulation, until the packet is received by the server. The context of the packet is actually deleted in **Server** class, in method *receiveStatus*. At this point we know that all CHs have received and finished processing the packet hence we can safely free the packet.

Note also that the same packet might be needed by clusterheads in future, in case if it interferes with a packet currently being received. Since each clusterhead decides upon this separately, it makes a copy of the packet when putting it in its interference list. This copying is done in **Clusterhead** object, in method *addInterference*. Therefore, when a packet is needed, a clusterhead will use the local copy as the global one might have been deleted by this time. This copy is deleted in method *deleteInterferenceList*, once the interference list is no longer needed.

Since in some comparison it may happen that a packet and its copy are compared, it is important not to compare pointers, but packets IDs, that can be obtained using *getId* method.

3.4 Node

Class **Node** implements a basic functionality of all nodes in the system, be it a sensor or a clusterhead. The two types of nodes, sensors and clusterheads, will be later derived from this class.

Node contains the following members:

- `static long id` - ID of the node. This ID is treated as the address and is set by the config file. We do not check uniqueness.
- `double x,y` - coordinates of the node.
- `Simulator *simulator` - the simulator to which the node belongs.

3.5 Sensor

Class **Sensor** inherits class **Node** and implements the basic sensor functionality. It implements the simples scheduling called instant access (IA). This means that the next available packet is scheduled as soon as the medium is free. It is latter derived to implement different types of scheduling implementations (RA and SA).

- `queue< Packet * > queue` - this is a queue where packets generated by sensing device are placed. Currently it has an unlimited size.

- `int busy` - the indicator is true if the sensor is currently transmitting a packet. That means that a newly arriving packet has to be placed in the queue rather than being directly transmitted.
- `int buf_size` - sensor buffer size. If a packet arrives when the queue is full, the packet will be dropped, and **PKT_DROP_Q** message will be generated.
- `Traffic *traffic` - contains a pointer to the traffic object that generates packet for this sensor.
- `void inQueue(Time now, Packet *p)` - if the queue is empty, schedules the packet straight away, otherwise stores it in the queue.
- `virtual void schedule(Time now)` - schedule the next packet transmission if the sensor is not already busy sending.

  The **Sensor** class itself implements instant access (IA). That means that the next available packet is scheduled immediately when the medium is free.

  The scheduling is done by inserting **PKT_START** event into the scheduler's queue at the current time instant (*now*).

  If the sensor is busy, *schedule* does not do anything, since it will be called latter anyway by *endTrans* method.
- `void startTrans(Time now)` - start actually transmitting the first packet in the queue on the medium. This method is called by scheduler when **PKT_START**. The role of *startTrans* is to schedule **PKT_END** event whose time depends on the size of the packet.
- `void endTrans(Time now)` - called by scheduler when the packet is transmitted (**PKT_END** event is executed). Since the transmitted packet is still in the queue, this method removes it from the queue and calls *schedule* to schedule the next packet.
- `void setTraffic(Traffic *t)` - sets the traffic generator to a sensor.

3.5.1 SensorIA

Performance of class **SensorIA** is actually implemented by **Sensor** class. See above for more details.

3.5.2 SensorRA

Class **SensorRA** inherits class **Sensor** and implements random access (RA) sensor nodes.

- `Time avg_delay` - average time of exponential backoff performed before sending each packet.
- `virtual void schedule(Time now)` - actually performs the exponential backoff. If the medium is idle, selects a random backoff and inserts the event **PKT_START** in the scheduler queue at that time. If the medium is not idle, does nothing.

3.5.3 SensorSA

Class **SensorSA** inherits class **Sensor** and implements scheduled access (SA) sensor nodes. Each node contains a time-hopping code, which is a set of several random numbers. The sensor performs backoffs using these numbers, each time the following one, looping to the first one when the last one is reached. This way a pseudo-random access is implemented such that the statistics are the same as for a random access, but clusterheads actually know schedules of sensors in advance.

In our implementation, a user needs to pass the average delay and the size of the time hopping code to the constructor. The code itself is generated at random by the constructor.

- `Time avg_delay` - average time of exponential backoff performed before sending each packet.
- `Time *delay` - an array of random numbers representing time-hopping codes.
- `int ds_size, ds_current` - total size of the time-hopping code, and the index of the current code being used.
- `virtual void schedule(Time now)` - performs the backoff equal to the current value of time-hopping code (delay[ds_current]).

3.6 Traffic

This class is responsible for generating new packets. One **Traffic** object is assigned to each sensor. At the startup, each sensor initializes its traffic object, which in turn generates the first packet. Whenever a packet is created within **Traffic** object (in other words, an event of type **PKT_CREATE** has occurred), it schedules the next packet by inserting an event of type **PKT_CREATE** in the scheduler's list at some point in time. This point is time defines the traffic statistics and is defined differently in different subclasses (exponential, uniform, etc.). Class **Traffic** is purely virtual (abstract) since it does not implement methods *next* and *printTraffic*.

- `Traffic(double p, long int ps, double me, int hd)` - constructor of the class. *p* is the TX power of a packet, *ps* is the size of an uncoded packet, *me* is the maximum error rate (defining the code) and *hd* denotes if packet contains real data. The last two parameters are by default set to 0. Note that the internal variable *packet_size* is not set to the uncoded packet size *ps* but is set to coded packet size obtained by dividing *ps* with code rate.
- `virtual void startAt(Time t)` - used to start a traffic generator. It will not generate a packet but will only insert the first event of type **PKT_CREATE** at time *t* in the scheduler. Then when this event is processed, the first packet will be generated (at time *t*), and the next packet will be scheduled.
- `Packet *genPacket(long int ps)` - generates a packet of size *ps* and returns a pointer to it. Currently does not do much. In future, some kind of coding should be done here.

- `virtual void next(Time t)` - this method is called at current time t. It should decide at what time t_1 the next packet p is to be send, and should call $schedAt(t_1, p)$. This method is called from *handleEvent* method of **Scheduler** class when handling **PKT_CREATE** event, to schedule the next packet.

  Method *next* is also responsible for generating packets. It generates a packet and passes it to *schedAt* method. This is done here since each traffic generator may generate packets of different sizes, transmitted powers and possibly contexts, and method *next* is meant to be the only one that has to be overloaded in an implementation of a specific traffic.

  This method is virtual and is not implemented here. It has to be implemented in inherited classes, and should reflect the chosen traffic distribution.

- `void printTraffic()` - prints the data about traffic generator. It is not implemented here and has to be implemented in the subclasses.

- `void schedAt(Time t, Packet *p)` - inserts a **PKT_CREATE** event in the scheduler for the next packet. Does not create the packet. The actual packet is generated in *next* method.

- `double getCodeRate()` - the rate of a code is calculated based on *max_error* (maximumm tolerable symbol error rate) under the assumption of BSC. For more details see Section 2.3.

- `long double power` - defines the transmitted power of the packet.

- `Sensor *sensor` - parent sensor of the traffic generator.

- `double max_error` - maximum symbol error rate that can occure without corrupting a packet from this source. Currently we implement only a simple model of coding. For more details see Section 2.3.

- `long int packet_size` - size of the packet (if it is fixed). This is a size of a coded packet. It is obtained by dividing the size of an uncoded packet *ps* with *getCodeRate*. For more details see Section 2.3.

- `int has_data` - defines if the generated packets should contain data

Change of traffic generator: ever sensor can change a traffic generator during a simulation. This is done by event **TRAFFIC_CHANGE**. When this event occurs, the scheduler will kill the existing traffic generator of a sensor, and set the new one, passed through **TRAFFIC_CHANGE** event. Events of type **TRAFFIC_CHANGE** are scheduled when parsing the configuration file.

3.6.1 TrafficExponential

Class **TrafficExponential** inherits **Traffic** class, and implements an exponential traffic generator. A time between two consecutive packets has exponential distribution with average *avg_delta*.

- `Time avg_delta` - average time between packets.

- `void next(Time t)` - uses exponential random generator to generate pause between two consecutive packets.

3.6.2 TrafficConstant

This is an implementation of a traffic generator that has regular patterns of both packet arrival times and packet sizes. Packet arrival times are stored in *delta_seq* and packet sizes in *packet_sizes*. For every new packet, next delay and the packet size are taken from these arrays in sequence.

- `Time *delta_seq` - sequence of packet inter-arrival times.
- `int ds_size` - length of *delta_seq* sequence.
- `int ds_current` - index to the next entry in *delta_seq* sequence to be used.
- `Time *packet_sizes` - sequence of packet sizes.
- `int ps_size` - length of *packet_sizes* sequence.
- `int ps_current` - index to the next entry in *packet_sizes* sequence to be used.

3.7 Clusterhead

Class **Clusterhead** defines a node that receives packets from sensors and tries to decode them. The goal of a clusterhead is to monitor packets on the wireless medium. If a clusterhead is free it can decide to try receives an incoming packet. Otherwise, it puts an incoming packet in the interference list, that contains all the packets that might interfere with its current reception.

We first briefly sketch the functionality of a clusterhead, and then we give a detailed information about each method. A clusterhead is notified about an arrival of a packet through *startTrans* method. There, the clusterhead has to decide if it wants to starts receiving that particular packet or ignore it. If ignored, the packet will be stored as a possible future interferer. At the end of the packet transmission, method *endTrans* is invoked. If it was the end of the packet being received, the actual demodulation is performed, and the result of the demodulation is passed to the central server.

- `list< PacketList > active` - lists all the packets currently being transmitted in the wireless medium. This includes the packet that is being received, if any.

  Whenever a sensor starts transmitted a packet, it invokes *startTrans* method of each clusterhead. In this method, the packet is then added to *active* list. When the transmission is over, a sensor calls *endTrans* method of a clusterhead, which removes the packet from *active* list.

- `list< PacketList > interfere` - lists all the packets that interfere with the packet currently being received. When a clusterhead starts a reception of a new packet, all packets from the *active* list are copied into the *interfere* list. If new transmissions start during the reception, these packets are added directly to the *interfere* list. Once the reception is finished, *interfere* list is emptied.

Note that the packet currently being received is in *interfere* list. This is due to the fact that a clusterhead may at any time switch to receive an another packet. When that happens, keeping always all the packets in *interfere* list facilitates clusterhead's implementation. However, that means that when calculating interference one should avoid including the packet being received (and remember here, packets are compared by their IDs, not pointers!).

- `Server *server` - a server to which the clusterhead is attached.
- `Channel *channel` - a channel model to be used by the clusterhead.
- `int startReceiving(Time now, Packet *p)` - true if the clusterhead should start receiving packet p. By default, a clusterheads wants to receive every packet. However, some adaptive threshold strategies might decide not to receive a packet if its power is to weak. See subclass' documentation (e.g. **ClusterNoncoherentSimple**) for more.
- `inline virtual int switchPacket(Time now, Packet *p)` - true if CH should switch to receiving p instead of currently receiving packet. By default, a clusterhead does not do that. However, in case of advanced architectures, such as switching ones, it might drop the packet being received in favor of a stronger one detected. See subclass' documentation for more.
- `DecodingResult *receive(Time now, Packet *p)` - returns the decoding result of packet p at a clusterhead. The actual demodulation/decoding staff is done here. Since the demodulation depends on a physical layer, the method is not implemented here and is purely virtual. In general, one should look at all the interfering packets from *interfere* list, and at the received packet and check the received SNR. See subclass' documentation (e.g. **ClusterNoncoherentSimple**) for more details.
- `void receiveNotification(int result)` - called by Server with the result of last packet decoding. Typically needed for adaptive mechanisms. Namely, if a clusterhead manages to decode a packet, it will update the parameters of an adaptive mechanism. However, in case of cooperative decoding, it may happen that clusterhead's decoding fails but that the packet is decoded at the server. In that case the server notifies the clusterhead about it using this method.
- `Time t_receiving` - arrival time of a packet currently being received.
- `Packet *p_receiving` - the packet being received.
- `void addInterference(Time t, Packet *p)` - adds packet p, arrived at time t to the interference list.
- `void deleteInterferenceList()` - empties the interference list. This is done when a packet packet reception is finished. It is also done in the destructor.
- `void startTrans(Time now, Packet *p)` - called whenever some sensor starts transmitting a packet. It is virtual, and is overloaded by some subclass to implement adaptive mechanisms.

- `DecodingResultType endTrans(Time now, Packet *p)` - called whenever some sensor ends transmitting a packet. It is virtual, and is overloaded by some subclass to implement adaptive mechanisms. Returns the result of decoding, but this returning argument is not further used at the moment.
- `void printActive()` - prints *active* list for debugging purposes.
- `void printInterference()` - prints *interfere* list for debugging purposes.
- `long double attenuation(Node *src)` - calculates interference from node *src* to the clusterhead.
- `long double delay(Node *src)` - calculates signal propagation delay from node *src* to the clusterhead.
- `Packet *getReceiving()` - returns *p_receiving*, that is the indicator whether the clusterhead is currently receiving a packet or not.
- `long double MRCcoef(Packet *p)` - returns a weighting coefficient (MRC or alike) a server should use for this clusterhead receiving packet p when combining it with other clusterheads. The method is purely virtual and is defined elsewhere.

There are several other types that are important for clusterhead functionality and we define them here:

- `PacketList` - for every packet in *active* and *interfere* list, we need to store not only the packet itself but also the starting time and the attenuation (since the packet class itself contains only the info about the transmitted power). Structure *PacketList* encapsulates all this data at one place.
- `DecodingResultType` - this is enumerated type that defines possible results of decoding. Possible results are:
  - NOT_LISTENING - the clusterhead did not decide to try to receive the packet
  - DECODE_ERROR - it tried to receive but failed
  - DECODE_OK - packet successfully received
  - DECODE_COOP - no clusterhead received the packet alone, but combining at server succeeded; used only by **Server** class)
  - SYNC_FAIL - bit error rate was so high that the clusterhead cannot even synchronize (which in turn means that this clusterhead cannot even further participate in combining).
- `DecodingResult` - class that contains different information about decoding process, including *DecodingResultType*. However, most of these information, unlike *DecodingResultType* are used solely by **Server** class, so we define it in Section 3.8.

3.7.1 ClusterGauss

Classes **ClusterGaussSimple**, **ClusterGaussSimpleOpt**, **ClusterGaussSwitch** are simple subclass of **Cluster** class that simulate a simple physical layer with Gaussian signals. All signals are assumed to be Gaussian, and a packet is received if the average SNR at the receiver during the transmission is higher than a given threshold. These implementations were done for testing purposes only and are not fully debugged.

3.7.2 ClusterNoncoherentSimple

This class is an subclass of **Cluster** class that implements physical layer described in [8]. Its main contribution is method *receive* that performs the demodulation described in [8]. This clusterhead also implements a fixed detection threshold, defined in *detect_thr*. If the received signal power of a packet is smaller than the threshold, the packet will not be received. File *cir.txt* is used to provide channel response samples, and it is necessary to provide this file if the class is used. The description of the file is given below.

- `long double detect_thr` - defines a fixed detection threshold. If the received signal power of a packet is smaller than the threshold, the packet will not be received.
- `virtual long double MRCcoef(Packet *p)` - returns a weighting coefficient (MRC or alike) a server should use for this clusterhead receiving packet p when combining it with other clusterheads. As shown in [8], if P is the packet received power at the clusterhead, T_i is the integration interval, B is the bandwidth and N_0 is the noise spectral density, then the MRC coefficient is $P/(T_i * B * N_0 + P)$.
- `virtual int startReceiving(Time now, Packet *p)` - true if the received power is higher then the threshold, meaning that the clusterhead should start receiving packet p.
- `virtual DecodingResult *receive(Time now, Packet *p)` - returns the result of decoding packet p at the clusterhead, using the physical layer from [8].
- `long double *CIR` - a pointer to channel impulse responses. They are loaded from a file called *cir.txt*. This is a plain text file which contains CIR_SIZE lines, each line containing CIR_SAMPLE numbers. A column in this file represents one measured channel impulse response. Resolution of these samples is defined in **ts_eff**, and is hard-coded at the moment. File *cir.txt* is parsed during construction of a clusterhead and loaded in the memory. Different CIR are latter used for different decodings.
- long double ts_eff - resolution of channel impulse responses (hard-coded at the moment).
- `void eval_interf(...)` - a private function that is called by *receive(...)* to help evaluating the decoding process.

Transmitted power of a packet: It is defined in *power* field of **Packet** class, and it represents energy of a pulse divided by 200 ns. For example, if a pulse energy is $15pJ$, then the value of *power* field should be 7.510^{-5}. This definition comes from the original physical model description where a time

slot was 200ns. The implementation of [8] still uses hard-coded time slot length of 200 ns. When one changes the bit rate of the channel, it only changes packet durations, and not the implementation of the decoding. This is, I believe, not perfectly correct since channel samples in `cir.txt` are done for 200ns time slots, but should be a fairly well and simple approximation.

3.7.3 ClusterNoncoherentAdaptive

This class is an subclass of **ClusterNoncoherentSimple** which implements the adaptive threshold strategy [5]. Each clusterhead first tries to get a list of all active sensors by listening to different packet. Every time a new sensor is discovered, its received power is put in *dt_list*. This ordered list is later used when adapting the threshold.

At the same time, a clusterhead monitors total load and utilization through the four variables: *util*, *utime*, *load* and *ltime*. Using these variables, a clusterhead decides whether and how to update the threshold. When a threshold is adapted, it is always adapted such that it includes or excludes one more sensor, using information about sensors from *dt_list*. The actual adaptation algorithm is described in [5], and the aforementioned variables are described in details below.

- `list< long double > dt_list` - an ordered list of different received powers. Each received power corresponds to one sensor (the actual ID of that sensor is not important). The information from the list is used when updating the threshold such that every time it is updated, exactly one node is excluded or included in the threshold region.
- `list< long double >::iterator current_dt` - a pointer to *dt_list* that points to the received power which corresponds to the actual value of the threshold. When the threshold is increased, it means that *current_id* is increased by one, and similarly decreased by one when the threshold is decreased.
- `long double utime` - the average estimated time between two received packets.
- `long double util` - the average utilization (1 / utime)
- `long double ltime` - the average estimated time between two detected packets.
- `long double load` - the average load (1 / ltime)
- `Time last_rcv` - the time of the last received packet (used to estimate utime).
- `Time last_end` - the time of the last detected packet (used to estimate ltime).
- `L_DOWN` - when the load is below this constant, the system is considered underutilized. Typically 60%.
- `L_UP` - when the load is above this constant, the system is considered over utilized. Typically 70%.
- `U_UP` - when the utilization is below this constant, the system is considered inefficient. Typically 20%.
- `ALPHA` - a constant that is used in the exponential weighted average filter

- `int count` - a counter that is increased every time the utilization is above the *U_UP* threshold, and is reset otherwise. It is used to update the threshold as described below.
- `N_THR_UP` - a constant that defines when a threshold should be decreased. If *counter* reaches `N_THR_UP` while the load is below *L_DOWN*, then the threshold will be decreased (more sensors admitted).
- `N_THR_DOWN` - a constant that defines when a threshold should be increased. If *counter* reaches `N_THR_DOWN` while the load is above *L_UP*, then the threshold will be increased (a fewer sensors admitted).
- `void startTrans(Time now, Packet *p)` - called at the beginning of a packet transmission. Does not implement any new code with respect to **ClusterNoncoherentSimple**.
- `DecodingResultType endTrans(Time now, Packet *p)` - called at the end of a transmission. If the decoding is successful, update *last_rcv*, *utime* and *util*. Also updates *last_end*.
- `virtual int startReceiving(Time now, Packet *p)` - called when a packet should be received. It first checks whether that sensor is already in the list of known sensors (that is if the received power is already in *dt_list*, since for us the attenuations are constant in time). Next it updates *load* and *util* as if we have just successfully received a packet. This is necessary in case when a threshold is high and a load had suddenly dropped; we then risk of not receiving any packets. Then, we update the threshold based on the current *load* and *util* estimates. Finally, we compare the received power to the threshold and decide whether or not to receive the packet.

3.7.4 ClusterNoncoherentSwitch

This class implement a switching clusterhead. The algorithm is simple: we always accept to receive a packet (therefore, we implement *startReceiving* to always return 1). If a stronger packet arrives, we switch receiving a stronger packet. This is implemented by overloading *switchPacket* accordingly. In the current implementation, there is no coding. That means that whenever two packet of similar powers overlap, there will be a collision. Therefore, we use a very simple heuristic: whenever a stronger packet comes that overlaps, we drop the weaker packet, even if the overlap is insignificant. If some kind of coding is introduced, this policy needs to be changed.

- `int startReceiving(Time now, Packet *p)` - when a receiver is idle, this method is called to decide whether we want to receive the incoming packet or nor. In this case, we always want to receive when idle, so it always returns 1.
- `int switchPacket(Time now, Packet *p)` - returns 1 if a clusterhead should switch from the packet currently being received to a newly arriving packet p. We always switch if the received power of p is higher.

3.8 Server

This class defines a central server. All clusterheads in the network are connected to the central server and forward information about received packets to it. In the simplest form, each clusterhead tries to decodes a packet on its own, and forwards the packet only if successfully decoded. The packet is received only if at least one clusterhead decodes it. This type of server is denoted as **SRV_SINGLE**.

More advanced servers allow clusterheads to combine the received information to perform a more successful decoding. One way to do so is to use majority decoding, which is denoted as **SRV_MAJORITY**. Each clusterhead sends a hard-bit sample (0 or 1) of each received symbol to the server. The server then simply decides upon every symbol depending on whether there are more 0s or 1s.

The second type of combined decoding is Maximum Ratio Combining (MRC). In this case, each clusterhead sends a soft sample of each received symbol to the server. These samples are weighted with clusterheads' weights (obtained through method *ClusterNoncoherentSimple::MRCcoef()*), and summed. The server then decides based on the resulting sum. This decoding is denoted as **SRV_MRC**.

A special class called **DecodingResult** is defined to carry the information passed from a clusterhead to a server. There are several objects of type **DecodingResult** for each packet. One object is created at the server, and one at each clusterhead. The server's object is initially empty, and a clusterhead's one carries the information about the decoding at that clusterhead. When a clusterheads finishes the decoding, it passes its **DecodingResult** object to the server. The server combines the received **DecodingResult** object with its own one, using *decodeOr()* method, described latter. Once the decoding results are received from all clusterheads, the server performs the actual combining decoding on the data it has received in its **DecodingResult** object.

Class **DecodingResult** has the following structure:

- `DecodingResultType result` - the information about the decoding at a clusterhead. It can be one of the following: **NOT_LISTENING, DECODE_ERROR, DECODE_OK, SYNC_FAIL**.
- `long double *data` - carries result of demodulation at the clusterhead for each symbol. In the case of majority decoding, it consists of -1s and 1s. In case of MRC, these are soft symbols. In case of no combining, this array is NULL.
- `int size` - size of *data* array.
- `Clusterhead *ch` - this is used for tracing purposes. When a clusterhead creates an **DecodingResult** object, it puts its own pointer here. The server, on the other hand, keeps in **DecodingResult** object a pointer to a clusterhead that first managed to decode the packet on its own. A pointer to the first clusterhead that sends **DecodingResult** containing **DECODE_OK** will be stored here. If the server does not received **DECODE_OK** message, this pointer will remain NULL. At the end of the day, if the overall result of the decoding is **DECODE_OK**, variable *ch* will contain a pointer to the clusterhead that managed to decode it. For a description of **DECODE_OK**, see a discussion in Section 3.7.
- `void decodeOr(DecodingResult *dr)` - this method is invoked from the server's **DecodingResult** object, with a clusterhead's **DecodingResult** object as a parameter. The goal is to update server's data on a packet with the information received from a clusterhead.

In case of a single-antenna approach, the combining is based on *result* field. If any of the results is **DECODE_OK**, the new value of the result field at the server will also be **DECODE_OK** (meaning that up to that moment, at least one clusterhead succeeded in decoding.) Otherwise, if any of the results is **DECODE_ERROR**, then the result will be **DECODE_ERROR** as well, meaning that no one decoded, but at least one clusterhead tried to decode and failed. Otherwise, if any of the results is **SYNC_FAILED**, then the result will be **SYNC_FAILED** as well, meaning that no one even managed to synchronize but at least one clusterhead tried to synchronize and failed. Finally, if nothing of the above is true, the result will be **NOT_LISTENING** meaning that none of the clusterheads even tried to receive the packet.

In case of multi-antenna approaches, in addition to the above, we also combine the received data. For each received symbol, the corresponding element of *data* array from the clusterhead's **DecodingResult** object is multiplied by a coefficient (1 in case of majority combining) and then added to *data* array of the server's **DecodingResult** object.

Another helper class, used by **Server** class is **PktInfo**. Whenever the server starts receiving informations about a packet, it creates **PktInfo** for that packet. The basic idea of this object is to keep the decoding information about the packet and to count from how many clusterhead we have received this information. Once the server receives it from all clusterheads, it can proceed to combined decoding if necessary, and decide if the packet is well received.

Class **PktInfo** has the following structure:

- `Packet *packet` - keeps a pointer to the packet being processed.
- `DecodingResult *status` - current status of the decoding of the packet (see above).
- `int noconf` - number of clusterheads that have send a decoding information about this packet until now.
- `void addCH(Clusterhead *ch, Time now, DecodingResult *s)` - a method called to add the decoding information from a clusterhead. It calls *decodeOr* method from *status*, and increments *noconf*.

Finally, we describe class **Server**. It has the following structure:

- `list< PktInfo * > pinfo` - a list of **PktInfo** objects for all packets currently being received by any of clusterheads.
- `Simulator *simulator` - a pointer to the simulator object.
- `ServerType type` - a type of the server. As already menitoned, server can be one of the following types: **SRV_SINGLE, SRV_MAJOR, SRV_MRC**.
- `void receiveStatus(Time now, Packet *p, Clusterhead *ch, DecodingResult *status)` - this method is called by a clusterhead to pass the decoding information about a packet to the server. It keeps a track of copies of packets received by different clusterheads. Once a copy of packet is received from each clusterhead, the method performs decoding by calling *decode* method (see below). Furthermore, if decoding is successful due to cooperation,

it invokes *receiveNotification* method of each clusterhead. This way clusterhead learn about a successful cooperative decoding and may chose to adapt threshold accordingly. Furthermore, *receiveStatus* method is responsible for deleting the working copy of a packet.

- `DecodingResultType decode(PktInfo *pi)` - performs the actual combining decoding if necessary. If the decoding result status is not **DECODE_ERROR**, it simply returns the status, as there is not much else a server can do. However, if the status is status is **DECODE_ERROR**, that means at least some clusterheads managed to synchronize to the packet, but none decoded it alone. In that case, the server tries the combining decoding, based on the linear combination of samples that is stored in **DecodingResult** object. If this decoding is successful, the status is changed to **DECODE_COOP**. Otherwise, it remains **DECODE_ERROR**.
- `PktInfo *genPacketInfo(Packet *p)` - generates a new **PktInfo** objects and returns a pointer to it. This function should be modified in case one wants to create an subclass of **Server** that will use an subclass of **PktInfo** class to carry data about a packet. It is not needed and not used at the moment.
- `void printResult(DecodingResultType res, Time now, Packet *p, Clusterhe *c)` - private method that prints trace information.
- `void transformResult(DecodingResult *s, Packet *p, Clusterhead *c)` - this method transforms the decoding result according to the combining policy. In case of single antenna server, it does not do anything. In case of majority combining, it transforms a soft sample into a hard sample (by using the sign function). Finally in case of MRC, it multiplies every sample with a weight obtained from the clusterhead. This method can be further extended to implement any linear multi-antenna technique.

3.9 Channel

This class contains information about parameters of the physical model and the wireless medium used in the simulation. It contains the following parameters:

- `long double a` - the attenuation power exponent
- `long double b` - the attenuation coefficient
- `long double n` - power spectral density of the background noise
- `long double rate` - rate of communication
- `long double distance(Node *src, Node *dst)` - calculates the distance between two nodes.
- `long double attenuation(Node *src, Node *dst)` - calculates the attenuation between two nodes. Currently, the attenuation is constant in time and depends only on the distance as the power law bl^a.
- `long double delay(Node *src, Node *dst)` - calculates the signal propagation delay between two nodes.

3.10 Simulator

This is a container object that contains the objects defined by the simulation scenario. This object parses a configuration file and create all objects defined therein. It creates the scheduler, insert the appropriate starting and ending events in the scheduler's queue and runs it. This class has the following structure:

- `Scheduler scheduler` - the scheduler that controls the timing of a simulation.
- `Server *server` - a pointer to the central server.
- `list< Clusterhead * > chs` - a list of all the clusterheads in the simulation.
- `list< Sensor * > sns` - a list of all the sensors in the simulation.
- `Channel *channel` - a pointer to an object that contains channel parameters.
- `void endAt(Time t)` - inserts an ending event in the scheduler's queue at time t. void run();
- `void parseConfig(const char *fname)` - parses the configuration file and creates the corresponding objects in the simulators. For details on the configuration file see Section 4.2.
- `void printConfig()` - prints the current configuration.

Chapter 4

Command Line, Configuration Files and Output

4.1 Command Line Parameters

To invoke the simulator, type:

```
./simulator <config> [<seed>]
```

The first parameter is a configuration file that describes the simulation scenario. It is described in details below. The second parameter is a seed.

If a seed is not specified, the simulator will take a random seed which is a function of time. Thus, the standard way to invoke the simulator is without seed, in which case the behavior will be purely random. Note that the random seed is a function of time, with precision of 1s, hence if you run short simulations more than once in a second, you might get identical results as the seeds will be the same. However, more frequently simulations last more than 1s so there is no need to worry about that.

Specifying a seed is useful for debugging purposes. In case of a bug, it is useful to replay exactly the same simulation. One can put any integer number as a parameter, and whenever the simulator is run with the same seed, it will produce the same output

There are two configuration files in the simulator. The first one described the simulation scenario, and is passed through the command line. The second one gives channel impulse response samples, to be used for a non-coherent receiver when simulating the packet reception. The name of the second one is hard-coded to `cir.txt` and has to be placed in the current directory.

4.2 Simulation Scenario Description

This file contains a description of the simulation scenario. It is passed to the simulator through the command line:

```
./simulator config.txt
```

The currently implemented parser is a very simple one. Each line is one command. If a line starts with # sign, it will be ignored (# has to be the first character in the line, that is, no space before

it!). Below, we give a list of configuration primitives with explanations of parameters. All units are standard (times are in seconds, powers (including noise power) are in Watts, packet sizes are in bits, rates are in bits per second).

- **Sensor:** a primitive to add a new sensor is

  ```
  Sensor <id> <x> <y> <type> <buf_size> <param>
  ```

  where <id> is the ID of the sensor (given by a user), <x>, <y> are the coordinates, and <type> denotes the type of the sensor and can be IA, which denotes instant access and creates an object of type **SensorIA**, RA which denotes random access and creates an object of type **SensorIA**, and SA which denotes scheduled access and creates an object of type **SensorIA**. buf_size defines the size of the sensor's packet queue.

  Parameter <param> denotes additional parameters that depend on the type. For a sensor of type IA there are no additional parameters.

  For the type <RA> the additional parameter is an average random access delay. Every transmission in this will be delayed by an exponential random time with the given average.

  For the type <SA> the additional parameters are a size of the code and an average random access delay. The sensor itself will create a random time-hopping code of given size, where each element will be a random exponential delay with the given average. The sensor will keep an index to the time-hopping code. Every transmission will be delayed by a value of the code pointed to by the index, and after the transmission the index will be moved to the next element (or to the first element if it has reached the end of the code). A typical size of the time-hopping code is 8.

- **Traffic:** a primitive to add a new traffic generator to a sensor is:

  ```
  Traffic <id> <sensor_id> <time> <type> <param>
  ```

  where <id> is the ID of the generator, <sensor_id> is the ID of the sensor to which this generator should be attached, <time> says at what time it should be attached (which will automatically disable the previous generator, if any), and <type> specifies which type of generator should be attached. Type of generator can be Constant or Exponential.

  In case of **constant** traffic generator, additional parameters are

  ```
  <power> <packet_distance> <packet_size> <max\_error> [Data]
  ```

  where <power> defines the transmitted power of packets <packet_distance> defines a constant timing between two packets, <packet_size> defines a constant packet size, max_error defines code (for more details see Section 2.3), and [Data] denotes that packets should carry a real data. If one simulates a Gaussian channel, there is no need to generate

the actual payload since the physical layer model does not need it (so [Data] is 0). In case of non-coherent 2-PPM UWB, described in [8], the underlying physical model needs actual data, so [Data] should be 1.

In case of **exponential** traffic generator, additional parameters are

```
<power> <packet_distance> <packet_size> <max\_error> [Data]
```

where <power> defines the transmitted power of packets <avg_packet_distance> defines the average time between two packets (which has an exponential distribution), <packet_size> defines a constant packet size, max_error defines code (for more details see Section 2.3), and [Data] denotes that packets should carry a real data.

- **Channel:** a primitive to define channel parameters is

```
Channel <a> <b> <N> <rate>
```

where fading on distance l is $b \times l^a$, <N> is the power spectral density of the white noise and <rate> is the channel nominal rate. Typical values for LOS channel attenuations, taken from [3] are $a = -1.7$ and $b = 1.38 \times 10^{-5}$. Typical white noise power, for a bandwidth of 1GHz, temperature $T = 300$K, and receiver noise figure 7dB is $N = 2.08 \times 10^{-11}$W (Boltzman constant is $K = 1.38 \times 10^{-23}$). Finally, for pulse energy of 30 pJ we have a channel symbol rate of 2500000 bits per second.

- **Server:** a primitive to define a central server is

```
Server <type>
```

where <type> can be one of: SingleUser, Majority, MRC.

- **Clusterhead:** a primitive to define a clusterhead is

```
Clusterhead <id> <x> <y> <type> <param>
```

where <id> is the ID of the clusterhead, <x>, <y> are the coordinates, and <type> is one of the following: **NoncoherentSimple, NoncoherentSwitch, NoncoherentAdaptive**. The first type has an additional parameter, <detect_threshold>, which defines a fixed detection threshold to be used (can be omitted or set to 0 if there is no detection threshold). The latter two types do not have additional parameters.

- **End:** a primitive used to stop the simulation is

```
End <time>
```

where <time> is the time at which the simulation will be stopped. This simply inserts an end event in the scheduler queue at time <time>.

4.3 Channel Impulse Response Samples

Channel impulse response samples file is called `cir.txt` and has to be placed in the current directory. This is a plain text file which contains CIR_SIZE lines, each line containing CIR_SAMPLE numbers. A column in this file represents one measured channel impulse response. Resolution of these samples is defined in **ts_eff**, and is hard-coded at the moment. File *cir.txt* is parsed during construction of a clusterhead and loaded in the memory. Different CIR are latter used for different decodings.

4.4 Simulator Output

During a simulation, the simulator outputs results to the standard output. Here we explain the content of this output. The first thing that is printed is the current configuration. This output summarizes the configuration file and is used primarily for debugging purposes. Here is an example:

```
********************************
Simulator configuration
********************************

Server: SingleAntenna

Channel: a=-3.300 b=-55.003[dB] N=-106.819[dB]

Sensor 1: x=4.000 y=4.000 ()
Sensor 2: x=1.000 y=1.000 ()

Clusterhead 1: x=5.000 y=5.000, type = NoncoherentSimple detect_thr= 0.000
```

In this example we have two sensors of type **Sensor** (that is with instant access), one clusterhead of type **ClusterNoncoherentSimple** and a single antenna server.

After printing a configuration, the simulator starts executing a simulation. During a simulation it displays a time and information about each event that occurs. Events are related to packets, and there are several types of them:

- **PKT_GENERATED** - a packet is generated. Prints out the packet ID and the source sensor's ID.
- **PKT_TRANS_START** - packet transmission started. When generated, a packet is stored in a sensor queue until scheduled for transmission. This events informs us when a sensor has actually started transmitting a packet. Note that the event is observed at the sensor side; different clusterhead will be aware of this event later, only after the signal propagates to them.
- **PKT_TRANS_END** - a packet transmission is finished and a sensor releases the medium. Note that the event is observed at the sensor side; different clusterhead will be aware of this event later, only after the signal propagates to them.
- **PKT_RCVD** - a packet is successfully received by at least one clusterhead (and there was no need for cooperative decoding). Prints out the packet ID, the source sensor's ID and the ID of

the clusterhead that first managed to decode the packet (if there are more of them, then the one closest to the sensor will be printed since the signal propagation is the shortest).

- **PKT_RCVD_COOP** - a packet has not been decoded by any single clusterhead but the cooperative decoding succeeded. The message is display at the moment when information about the packet are gathered from all the clusterheads.
- **PKT_ERROR** - a packet has not been decoded, but at least one clusterhead has synchronized to it and try to decode.
- **PKT_SYNC_FAIL** - at least one clusterhead was trying to synchronized to a packet, but none has succeeded in synchronizing.
- **PKT_DROP** - a packet is dropped as no clusterhead has decided to try to receive it.
- **PKT_DROP_Q** - a packet is dropped at the sensor queue since the traffic on sensor was to high and queue is overfull.

The format of the trace output is defined in `trace.h` file. We next give an example of the output with explanations:

```
0.766351205785 PKT_GENERATED   pkt: 165 sns: 1
0.768701848415 PKT_TRANS_START pkt: 158 sns: 1
0.768703848415 PKT_TRANS_END   pkt: 158 sns: 1
0.768703985690 PKT_RCVD        pkt: 158 sns: 1 ber: 0.000e+00 ch: 2
0.769164785933 PKT_GENERATED   pkt: 166 sns: 1
0.769190400153 PKT_TRANS_START pkt: 159 sns: 1
0.769192400153 PKT_TRANS_END   pkt: 159 sns: 1
0.769192537428 PKT_DROP        pkt: 159 sns: 1 ber: 1.000e+00
0.771468494639 PKT_TRANS_START pkt: 160 sns: 1
0.771470494639 PKT_TRANS_END   pkt: 160 sns: 1
0.771470631914 PKT_SYNC_FAIL   pkt: 160 sns: 1 ber: 1.752e-01
0.777236021990 PKT_TRANS_START pkt: 161 sns: 1
0.777238021990 PKT_TRANS_END   pkt: 161 sns: 1
0.777238159265 PKT_ERROR       pkt: 161 sns: 1 ber: 1.875e-02
0.777729482645 PKT_TRANS_START pkt: 162 sns: 1
0.777731482645 PKT_TRANS_END   pkt: 162 sns: 1
0.777731619920 PKT_RCVD_COOP   pkt: 162 sns: 1 ber: 0.000e+00
0.777734152174 PKT_GENERATED   pkt: 167 sns: 1
0.781251181569 PKT_GENERATED   pkt: 168 sns: 1
```

From the log we see that packets 165 to 168 were generated, packets 158 to 162 were transmitted, packets 158 was received by clusterhead 2 alone (with no bit errors), packet 159 was dropped because no one decided to try to receive it, synchronization to packet 160 failed (bit-error rate was 1.752×10^{-1} which is above 10%), packet 161 is lost since not even cooperative decoding helped receiving it (bit-error rate was 1.875×10^{-2} and no coding was used), and finally packet 162 was received by means of cooperative decoding (again with no bit errors).

Chapter 5

Future Work

Here is a list of some features that should be implemented in the simulator in future versions:

- **Packet coding:** currently, only a simple model of packet coding, described in Section 2.3 is implemented. In order to have more realistic coding performance, on should implement a real coding and use it to test packet error rate.

Bibliography

[1] J. Barry, D. Messerschmitt, and E. Lee. *Digital Communication: Third Edition*. Kluwer Academic Publishers, 2003.

[2] T. Cover and J.A. Thomas. *Elements of Information Theory*. John Whiley & Sons, 1991.

[3] S.S. Ghassemzadeh and V. Tarokh. Uwb path loss characterization in residential environments. In *IEEE Radio Frequency Integrated Circuits (RFIC) Symposium*, pages 501–504, June 2003.

[4] F.P. Kelly, A.K. Maulloo, and D.K.H. Tan. Rate control in communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research Society*, 49:237–252, 1998.

[5] B. Radunović and H.L. Truong. On architectures of transmit-only wireless sensor networks. *Technical Report*, 2004.

[6] D. Rus. Keynote on autonomous mobile networks. In *The First IEEE Workshop on Embedded Networked Sensors (EmNetS-I)*, 2004.

[7] B. van der Wal and al. D2a2 - definition of uwb scenarios. *Technical Report from PULSER WP2*, 2004.

[8] M. Weisenhorn. Physical layer for reader scenario. *IBM Technical Report*, 2004.

Appendix A

Sample Script Files

In this section we give examples of script files that are used to generate random scenarios and invoke the simulator. We also give an example how to parse the output results. We give two scenarios, a static and a dynamic one.

A.1 Static Scenario

In a static scenario, nodes are randomly placed on a square, and several clusterheads are regularly placed on the same square. We vary different parameters, and for each set of parameters we run several simulations. We print all the results in a file `log.txt` which can latter be used from MATLAB to plot the results.

A.2 script.pl

```
#!/usr/bin/perl -w
use strict;

$| = 1;

#The optimal average backoff is 20 for 20 nodes.
# as the rule of a tumb, one canuse ns/2

my @nsn = (2,5,10,20,50);
my @nch = (2);
my @stype = ("IA", "RA", "SA");
my @ctype = ("NoncoherentAdaptive", "NoncoherentSimple", "NoncoherentSwitch");
my @srtype = ("SingleUser", "MRC");
my @powers = (1.5e-4);
my @avgds = (0.8000, 0.1600, 0.0400, 0.0160, 0.0080,
             0.0040, 0.0027, 0.0020, 0.0016);
my @avgsds = (1e-4);
my @dthrs = (0);
my @coders = (0, 0.05, 0.1);
my $sim_length = 1;
my $no_packet = 200;
```

```
my $ps = 800;
my $run = 5;
my $size = 40;


# here we'll keep distances of all sensors
# from the central point (defined in make_config)
my @distances = ();


my ($r, $ns, $nc, $st, $stn, $ct, $srt, $ctn, $srtn, $power,
    $avg_delay, $avg_sch_delay, $det_thr, $coder);

$det_thr = 0;




# Function make_topo makes a random distribution of sensor nodes. It
# is separated from make_config so that we can use the same topology
# with different network parameters (e.g. clusterhead architecture)
# and have a meaningful performance comparison independent of topology
# variance. The function stores the topology into topo.txt file. The
# topology is latter copied from that file into config.txt file.

sub make_topo {
    my ($i, $x, $y, $lns, $lnc, $lsim_length, $lst, $lct, $lsrt, $lpower,
        $lavg_delay, $lavg_sch_delay, $lps, $ldet_thr, $lsize, $lcoder,
        $c1, $c2, $c3, $tstart, $cx, $cy);

    ($lns, $lnc, $lsize, $lst, $lavg_sch_delay) = @_;


    my $out="topo.txt";
    open OUT, ">$out" or die "Cannot open $out for write :$!";


    # we want to have a 40m x 40m
    my $lxsize = $lsize;
    my $lysize = $lsize;



    # define the central point coordinate from which we calculate the distance
    $cx = $size/2;
    $cy = $size/2;


    foreach $i (1..$lns){
        $x = rand($lxsize);
        $y = rand($lysize);

        push(@distances, sqrt(($x-$cx)**2 + ($y-$cy)**2));
```

```
        if ($lst eq "IA") {
            print OUT "Sensor $i $x $y 100 IA\n";
        } elsif ($lst eq "RA") {
            print OUT "Sensor $i $x $y 100 RA $lavg_sch_delay\n";
        } else {
            print OUT "Sensor $i $x $y 100 SA 8 $lavg_sch_delay\n";
        }

        # Traffic generators are attached in make_config
    }

    close OUT;
}



# The main configuration function. It creates the server, clusterheads, channel
# and traffic objects. It copies sensors topology from topo.txt file.

sub make_config {
    my ($i, $x, $y, $lns, $lnc, $lsim_length, $lst, $lct, $lsrt, $lpower,
        $lavg_delay, $lavg_sch_delay, $lps, $ldet_thr, $lsize, $lcoder,
        $c1, $c2, $c3, $tstart, $cx, $cy);

    ($lns, $lnc, $lsim_length, $lst, $lct, $lsrt, $lpower, $lavg_delay,
     $lavg_sch_delay, $lps, $ldet_thr, $lsize, $lcoder) = @_;


    # DEBUG - to avoid singularities, we schedule
    $lavg_sch_delay = $lavg_delay;

    my $out="config.txt";
    open OUT, ">$out" or die "Cannot open $out for write :$!";


    # Deduce rate from power such that we reach the limit
    my $crate = 5e6 * 7.5e-5 / $lpower;

    print OUT "Server  $lsrt\n";
    #NLOS
    #print OUT "Channel -3.3 3.16e-6 2.08e-11 $crate\n";
    #LOS
    print OUT "Channel -1.7 1.38e-5 2.08e-11 $crate\n";


    # define the central point coordinate from which we calculate the distance
    $cx = $size/2;
    $cy = $size/2;

    if ($lnc == 1) {
        $c1 = $size/2;
        if ($lct eq "NoncoherentSimple") {
            print OUT "Clusterhead 1 $c1 $c1  NoncoherentSimple $ldet_thr\n";
```

```
    } elsif ($lct eq "NoncoherentSwitch") {
        print OUT "Clusterhead 1 $c1 $c1  NoncoherentSwitch \n";
    } else {
        # DEB
        print OUT "Clusterhead 1 $c1 $c1  NoncoherentAdaptive $lavg_delay\n";
    }
} elsif ($lnc == 2) {
    $c1 = $size * 1./2;
    $c2 = $size * 1./4;
    $c3 = $size * 3./4;
    if ($lct eq "NoncoherentSimple") {
        print OUT "Clusterhead 1 $c2 $c1  NoncoherentSimple $ldet_thr\n";
        print OUT "Clusterhead 2 $c3 $c1  NoncoherentSimple $ldet_thr\n";
    } elsif ($lct eq "NoncoherentSwitch") {
        print OUT "Clusterhead 1 $c2 $c1  NoncoherentSwitch \n";
        print OUT "Clusterhead 2 $c3 $c1  NoncoherentSwitch \n";
    } else {
        print OUT "Clusterhead 1 $c2 $c1  NoncoherentAdaptive \n";
        print OUT "Clusterhead 2 $c3 $c1  NoncoherentAdaptive \n";
    }
} elsif ($lnc == 3) {
    # TBD!
} else {
    $c1 = $size * 1./4;
    $c2 = $size * 3./4;
    if ($lct eq "NoncoherentSimple") {
        print OUT "Clusterhead 1 $c1 $c1  NoncoherentSimple $ldet_thr\n";
        print OUT "Clusterhead 2 $c1 $c2  NoncoherentSimple $ldet_thr\n";
        print OUT "Clusterhead 3 $c2 $c1  NoncoherentSimple $ldet_thr\n";
        print OUT "Clusterhead 4 $c2 $c2  NoncoherentSimple $ldet_thr\n";
    } elsif ($lct eq "NoncoherentSwitch") {
        print OUT "Clusterhead 1 $c1 $c1  NoncoherentSwitch \n";
        print OUT "Clusterhead 2 $c1 $c2  NoncoherentSwitch \n";
        print OUT "Clusterhead 3 $c2 $c1  NoncoherentSwitch \n";
        print OUT "Clusterhead 4 $c2 $c2  NoncoherentSwitch \n";
    } else {
        print OUT "Clusterhead 1 $c1 $c1  NoncoherentAdaptive \n";
        print OUT "Clusterhead 2 $c1 $c2  NoncoherentAdaptive \n";
        print OUT "Clusterhead 3 $c2 $c1  NoncoherentAdaptive \n";
        print OUT "Clusterhead 4 $c2 $c2  NoncoherentAdaptive \n";
    }
}


# Copy SNs topology from a precreated topo.txt file

open TOPO, "topo.txt";

while (<TOPO>) {
    print OUT "$_";
}


# Attach traffic generators
```

```
    foreach $i (1..$lns){
        # do not start all traffic at 0 since this confuses the estimators!
        # start them with a delay with an exp. distribution
        $tstart = -$lavg_delay * log(1. - rand(1));
        print OUT "Traffic $i $i $tstart Exponential ";
        print OUT "$lpower $lavg_delay $lps $lcoder Data\n";
    }



    # Ending time

    print OUT "End     $lsim_length\n";
    close OUT;
}




# This subroutine calculates different staistics.
# For each sensor we monitor how many packets it has generated,
# and how many it has successfully transmitted. We also track
# the distance to the farmost sensor that successfully transmitted
# a packet (called the range) and we track when will 90% of sensors
# manage to convey any packet (called the outage time).


sub calc_stat {
    my ($ns, @distances);
    ($ns, @distances) = @_;

    # We return number of rcvd packets, fraction of rcvd packets, utility,
    # time until 90% of sensors receive a packet, and the maximum range

    my ($sum, $util, $fsum, $ftotal, $futil, $otime, $range);
    my $no_sn_rcvd = 0;
    my $hrd_sns = 0;
    $range = 0;
    $otime = 0;
    #DEB
    $sum = 0;

    # Here we keep the list of all sensors and the number of sent and
    # received packet from eah one so that we can calculate stats at the end.
    my @sensorID = ();
    my @gendata = ();
    my @rcvdata = ();
    my $i;

    open LOG, "tmp.txt" or die "Cannot open tmp.txt for read :$!";

    while (<LOG>) {
```

```
if ($_=~/PKT_GENERATED/){
    my @res = split /\s+/, $_;
    my $time = $res[0];
    my $action = $res[1];
    my $pkt_id = $res[3];
    my $sn_id  = $res[5];

    # if a packet is generated by a sensor which we don't know
    # yet, we have to store the sensor data in arrays
    # sensorID, gendata and rcvdata. We do that here.

    $i = 0;
    while ($i <= $#sensorID && $sensorID[$i] != $sn_id) {
        $i++;
    }

    if ($i > $#sensorID) {
        push(@sensorID, $sn_id);
        push(@gendata, 1);
        push(@rcvdata, 0);
    } else {
        $gendata[$i]++;
    }

}

if ($_=~/PKT_RCVD/ || $_=~/PKT_RCVD_COOP/){
    my @res = split /\s+/, $_;
    my $time = $res[0];
    my $action = $res[1];
    my $pkt_id = $res[3];
    my $sn_id  = $res[5];


    # register that sensor $sn_id has received a new packet
    $i = 0;
    while ($i <= $#sensorID && $sensorID[$i] != $sn_id) {
        $i++;
    }

    if ($i > $#sensorID) {
        print "@sensorID\n";
        print "$sensorID[$#sensorID] $sensorID[0]\n";
        die "Error: no sensor $sn_id!";
    } else {
        # First packet for this sensor -> increase hrd_sns
        if ($rcvdata[$i] == 0) {$hrd_sns++;}
        if ($otime == 0 && $hrd_sns * 1. / $ns > 0.9) {$otime = $time;}
        # Add one to the number of packet received by this sensor
        $rcvdata[$i]++;
    }

    # check if that is the largest range by now
    if ($distances[$sn_id-1] > $range) {
```

```
                $range = $distances[$sn_id-1];
            }

        }
    }

    # here we add 0.1 to every log() to avoid -inf
    $util = 0;
    $futil = 0;
    $sum = 0;
    $fsum = 0;
    $ftotal = 0;
    for $i (0 .. $#sensorID) {
        $util   = $util + log(0.1 + ($rcvdata[$i] * 1.)/$sim_length);
        $futil  = $futil + log(0.1 + ($rcvdata[$i] * 1.)/$gendata[$i]);
        $sum    = $sum + $rcvdata[$i] * 1./ $sim_length;
        $fsum   = $fsum + $rcvdata[$i];
        $ftotal = $ftotal + $gendata[$i];
    }

    $fsum = $fsum * 1. / $ftotal;
    ($sum, $fsum, $util, $futil, $otime, $range);
}




my $res="log.txt";
open RES, ">$res" or die "Cannot open $res for write :$!";


foreach $avg_delay (@avgds) {
  foreach $avg_sch_delay (@avgsds) {
    foreach $ns (@nsn) {
      foreach $nc (@nch) {
        foreach $st (@stype) {
          foreach $r (1..$run) {

            # Here we want to use the same network topology for
            # different parameters, so that the performance difference
            # does not depend on topology difference.  That's why we
            # first generate the topology (sensor coordinates) and
            # then use the same one for different power, coder, ctype
            # and srtype.

            make_topo($ns, $nc, $size, $st, $avg_sch_delay);

            foreach $power (@powers) {
              foreach $ct (@ctype) {
                foreach $srt (@srtype) {
                  foreach $coder (@coders) {

                    # We want each sensor to generate on average
```

```
                # no_packet packets In statistics we already
                # divide number of packets by the simulation time
                # to get the rate (in pack/s)
                $sim_length = $avg_delay * $no_packet;

                make_config($ns, $nc, $sim_length, $st, $ct, $srt,
                            $power, $avg_delay, $avg_sch_delay,
                            $ps, $det_thr, $size, $coder);
                system("./simulator config.txt>tmp.txt") == 0
                       or die "Command failed!";

                my ($sum, $fsum, $util, $futil, $otime, $range) =
                       &calc_stat($ns, @distances);

                if ($st eq "IA") {
                  $stn = 0;
                } elsif ($st eq "RA") {
                  $stn = 1;
                } else {
                  $stn = 2;
                }

                if ($ct eq "NoncoherentSimple") {
                  $ctn = 0;
                } elsif ($ct eq "NoncoherentAdaptive") {
                  $ctn = 1;
                } else {
                  $ctn = 2;
                }

                if ($srt eq "SingleUser") {
                  $srtn = 0;
                } elsif ($srt eq "Majority") {
                  $srtn = 1;
                } else {
                  $srtn = 2;
                }

                print "$ns $nc $st $ct $srt $power $avg_delay "
                print "$avg_sch_delay $det_thr $coder $r $ps $size";
                printf ("\t %ld %.4e %.4e %.4e %.4e %.4e\n",
                       $sum, $fsum, $util, $futil, $otime, $range);

                print RES "$ns $nc $stn $ctn $srtn $power $avg_delay ";
                print RES "$avg_sch_delay $det_thr $coder $r $ps $size";
                print RES "\t $sum $fsum $util $futil $otime $range\n";

                system("sleep 1");
              }
            }
          }
        }
      }
    }
```

```
      }
    }
  }
}
```

A.3 plot_cs.m

Here we give a simple MATLAB script that produces the graphical output of results. There are three preselected columns, X, Y and Z. We first find all the distinct values in z column. For example if Z column denotes a type of clusterhead (NoncoherentSimple - 0, NoncoherentAdaptive - 1, NoncoherentSwitch - 2), then the program will first select all record with Z=0 and plot the results for NoncoherentSimple, then select Z=1 and plot, etc.

When one set of Z has been selected, then the program loops over distinct values of X. For each value of X it finds all the corresponding values of Y (there are several, depending on how many independent runs we did) and calculates the mean and the confidence interval. Finally, it plots this.

```
log = load('log.txt');

xcol = 7;                   % which column to be displayed on x axis
ycol = 14;                  % which column to be displayed on y axis
zcol = 10;                  % column to be filtered
zcoln = 'code';             % name of Z column (to be displayed in legend)

% preselect which scenario do we choose
nch = 1;                    % number of clusterheads
nsn = 50;                   % number of sensors
ssize = 40;
ps = 800;
chtype = 0;
srvtype = 0;


%%% read simulation parameters

R = unique(log(:,11));
norun = length(R);

pow = unique(log(:,6));
if length(pow) > 1
  ppp
end

rate = 5e6 * 7.5e-5 / pow;
sym_len = 1/rate;

rate = rate / 1e6;       % mbps
pdur = ps * sym_len;

% select distinct Z
Z = unique(log(:,zcol)');
```

```
X = unique(log(:,xcol)');
Xp = ps ./ X * nsn / 1e6;


hold on;


sty = {'-k', '-.b', '--g', ':r', '-g', '-.r',...
       '--k', ':b', '-b', '-.k', '--r', ':g'};
lp = [];
str = {};

for z = Z
  ind = find(z==Z);

  me = zeros(size(X));
  sv = zeros(size(X));

  for i=1:length(X)
    r = log(find(log(:,2) == nch),:);
    r = r(find(r(:,1) == nsn),:);
    % preselect Size
    r = r(find(r(:,13) == ssize),:);
    % preselect packet size
    r = r(find(r(:,12) == ps),:);
    % preselect NoncoherentSimple
    r = r(find(r(:,4) == chtype),:);
    % preselect MRC
    r = r(find(r(:,5) == srvtype),:);
    % preselect DT=0
    r = r(find(r(:,9) == 0),:);
    r = r(find(r(:,zcol) == z),:);
    r = r(find(r(:,xcol) == X(i)), ycol) * ps / 1e6;
    me(i) = mean(r);
    sv(i) = std(r);
  end


  lineprop = errorbar(Xp, me, sv, sty{ind});
  lp = [lp, lineprop];
  str{ind} = sprintf('%s = %.2f', zcoln, z);
end



hold off;


fs = 16;
set(gca, 'FontSize', fs);

%xlim([Xp(1) Xp(length(Xp))*1.1])
xlim([0 Xp(1)*1.01])
yl = ylim;
```

```
ylim([0 yl(2)])

xlabel('Aggregate input rate [Mbps]', 'FontSize', fs);
ylabel('Aggregate effective output rate [Mbps]', 'FontSize', fs);

if chtype == 0
  sCH = 'Simple';
elseif chtype == 1
  sCH = 'Adaptive';
else
  sCH = 'Switch';
end

if srvtype == 0
  sSRV = 'SA';
else
  sSRV = 'MRC';
end


title(sprintf('#SN = %d, #CH = %d, size = %d, %s, %s', ...
      nsn, nch, ssize, sCH, sSRV));

[legend_h, object_h] = legend(lp, str, 1);
set(object_h(1),'FontSize',fs);
```