Chapter 1

Introduction

1.1 Introduction

In the last decade, there is a rapid growth of wireless sensor network. It is the network consisting of many small, low-power, and intelligent sensor nodes and one or more base stations. Wireless sensor network consists of three main things called nodes (or motes), a processor and a radio transceiver. A sensor node is a battery-powered device that is equipped with sensors such as seismic, light, temperature, and acoustic. A processor for mapping a physical quantity taken from the environment to a quantitative measurement as well as performing network protocol functions and a radio transceiver for communication. Figure 1.1 shows the simple wireless sensor network.

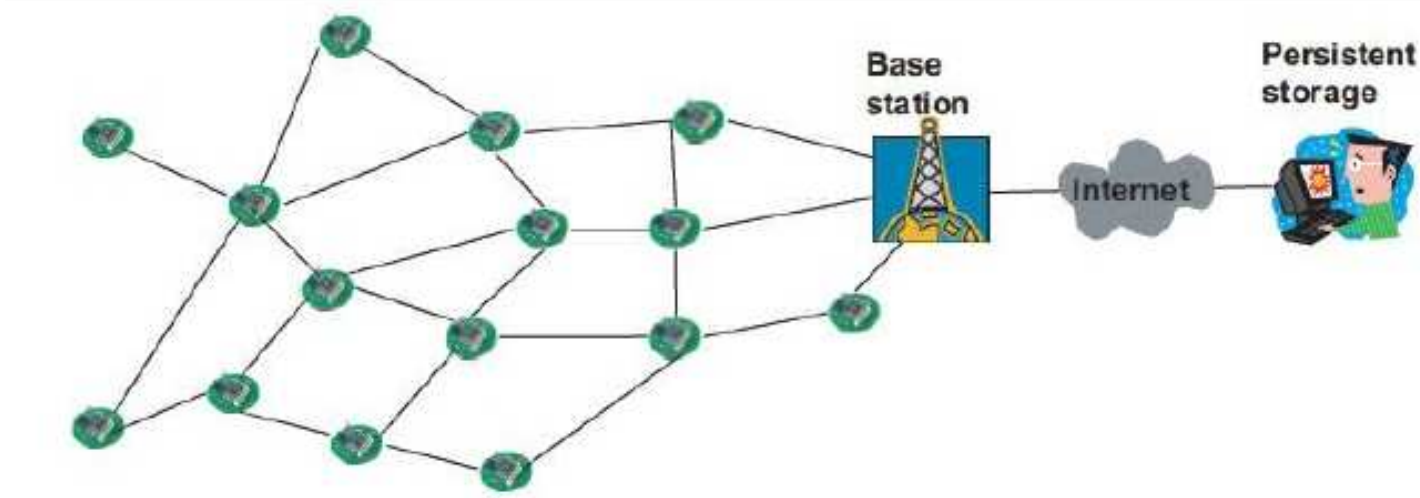

Fig 1.1 wireless sensor network

Sensor nodes are placed in a physical area of interest, for example, natural ecosystems, battlefields, and man-made environments to gather high-level description of events such as temperature, sound, motion, vibration or pressure from the environment and send data to the base station, through which an end user retrieves the data. The base station, which may be a laptop or wireless device, acts as a gateway between sensor nodes and the end user.

Wireless sensor network's applications can be categorised according to their operational paradigm: data gathering and event-driven. Data gathering applications require sensor nodes to periodically report their data to the base station [3]. Sensor nodes serve as both gatherers and routers for the data. Examples of data gathering applications are as follows.

- Temperature and humidity monitoring in the canopy of potato plants for precision agriculture.
- Habitat monitoring at Great Duck Island and microclimate monitoring in redwood trees.

In the event-driven paradigm, sensor nodes only send data when an event of interest occurs. Examples of event driven applications are as follows.

- Sensor nodes are used to navigate users through the monitored environment in the presence of hazardous events (e.g., a chemical spill, a traffic accident).
- To track a fire as it spreads and form a perimeter for the fire area.
- To follow the movement of a source to be tracked (e.g., object tracking, trespasser detection).
- To detect intrusion in the sand and to monitor vibration (e.g., wind and earthquakes) that could damage the structure of a building.

A sensor node works under severe resource constraints such as limited battery power, computing power, memory, wireless bandwidth and communication capability, while the base station has more computational, energy and communication resources. Sensor nodes are typically deployed in a remote location. It is expensive in terms of cost and effort and often impractical to replace their batteries. Sensor nodes must operate efficiently in order to prolong the network lifetime and so make sensor network applications economically feasible. Existing research in wireless sensor networks has focused largely on the energy efficiency aspect of sensor networks. Following are some of the challenges for designing sensor network protocols.

- First challenge is to design an energy-efficient protocol that maintains good network performance. The most common approach to conserve energy is to alternate sensor node duty cycles between high power active state and low power sleep state. MAC protocols typically trade off performance for a decrease in energy consumption to prolong a sensor node's lifetime .However, as new applications of sensor networks emerge, classical network performance parameters such as latency, throughput and reliable data delivery, gain importance. For example, latency assurance is important in wireless sensor network applications such as factory monitoring, vehicle tracking and detection. Throughput

guarantee of data gathered by sensor nodes is crucial in smart homes video surveillance in order to achieve optimum benefits of such application. The challenge is to find an optimal solution that provides a balance between energy efficiency and network performance.

- Second challenge is to design a traffic adaptive protocol that can learn the current channel conditions and adapt themselves accordingly in order to reduce transceiver energy consumption. In most sensor network applications, traffic loads in different parts of the networks can vary significantly over time. For example, a sensor network may start with light traffic where nodes only respond to a query disseminated to the network and when nodes in certain part of the network detect an event of interest, that part of the network begins rapid data gathering tasks. A protocol that works well under low contention traffic, but is unable to adapt to changing traffic patterns, may consume more energy than necessary and so decrease network lifetime. Following the previous example, sensor nodes in this type of protocol often return to the idle listening phase after they have finished reporting an event. However, adaptation often includes complexity and protocol designers should consider how that complexity affects the processing and memory resources of sensor nodes.
- The last challenge is to allow protocols to adapt to changes in network topologies in order to conserve energy, maintain network performance and maintain the fine-granularity of information gathered from the networks. Although in most sensor network applications, nodes are designed to be stationary, these nodes may suffer temporary or permanent failures due to external factors or energy depletion, and new nodes may be added to the network at any time. A protocol designed for sensor networks should be able to detect these network dynamics and function properly without degrading the network performance.

These challenges and observations lead to the design of a flexible-schedule-based TDMA Protocol (FlexiTP). FlexiTP is distinguished from existing energy-efficient protocols by providing end-to-end guarantees on data delivery within energy and memory constraints of wireless sensor networks. FlexiTP also adapts to traffic fluctuations and topology changes. It is

also scalable for large number of nodes. By restricting the number of children, the energy consumption per node during data gathering reduces.

1.2 Motivation

The field of sensor networks is growing rapidly now days. Also, we have studied Data Communication and Networking course which made easy to understand this field. This project contains the communication part as well as software part (C++ language). So, this was the best way to go into and learn two different areas simultaneously. These motivated us to do this project.

1.3 Scope

There are seven different layers in the OSI (Open Systems Interconnection) model namely physical, data link, network, transport, session, presentation and application layer. Data link layer does routing as well as medium access control. These two tasks are very important for any network protocols. These protocol works at data link layer for routing and medium access control. It also works at application layer for time synchronisation.

1.4 Objective

The objective of this project is to first learn and simulate the S-MAC protocol. Here we have simulate both S-MAC and IEEE 802.11 protocol and compare these two protocol.

Outline of thesis

This thesis is organised as follows.

In chapter 2, the working and theory related to the S-MAC.

In chapter 3, the Network Simulator, namtrace, trace format, awk file and Gnuplot is discussed.

In chapter 4, simulation of S-MAC in NS2 are discussed.

In chapter 5, simulation result will be displayed.

In Chapter 6, conclusion of this thesis will be done.

Chapter 2

S-MAC Protocol

2.1 Overview of S-MAC

S-MAC, the abbreviation for Sensor-MAC, is a medium access control (MAC) protocol designed for wireless sensor networks, proposed by SCADDS project group at USC/ISI [11]. Wireless sensor networking technology is emerging in these years and becomes a popular research area of computer science and technology. Wireless sensor networks are the integration of sensor techniques, nested computation techniques, distributed computation techniques and wireless communication techniques. They can be widely used in many areas, such as environment monitoring, medical systems, and intelligent building systems. The features for wireless sensor networks are listed as below:

- Such networks consist of many distributed sensor nodes. Each node has one or more sensors, an embedded processor, and a low-power radio, and is normally battery-operated.
- In some applications, sensor nodes are distributed within a vast expanse, such as earthquake monitoring in desert area. Sometimes their working environments are quite dangerous and even not accessible for humans.
- Normally, sensor nodes in one network collaborate for a common application. They communicate and exchange information in a peer-to-peer way (ad hoc fashion), instead of by accessing a base station.
- Sensor nodes keep silent for most of the time, but they will become active suddenly when something is detected.
- As time elapses in a network, some nodes may die, some nodes may move away, some new nodes may be added. The topology may change over time.
- A message is a meaningful unit of data that a node can process. For saving energy, messages will be processed in a store-and-forward fashion, which is called in-network data processing, instead of being processed at a certain node in a centralized way.

From these features, we can identify the following problems that we have to solve, when we design a good MAC protocol for wireless sensor networks.

1. The foremost one is energy efficiency.

As stated above, sensor nodes are expected to be battery-equipped. Due to their working environments, recharging or replacing batteries for each node is difficult and uneconomical, sometimes even impossible. Therefore, how to reduce the energy consumption to prolong the service lifetime of sensor nodes becomes a critical issue. To solve the energy problem, we should find out the sources of energy waste. As viewed from hardware layer, radio communication consumes most of energy. Moreover, the usage of radio has close relation with MAC protocols. Among existing wireless MAC protocols for shared-medium networks, we have identified the following major sources of energy waste.

- The first one is idle listening. For contention-based MAC protocols, such as IEEE 802.11 ad-hoc mode, in order to perform effective carrier sense against possible collisions, it puts nodes to listen to the channel all the time when nodes are idle. And radio will consume almost the same power as in receiving state. A considerable percentage of energy will be wasted on idle listening, especially when the traffic load on the network is light. Among those factors for energy waste, idle listening is a dominant one.
- The second one is collision or corruption. Normally, collision may occur when neighboring nodes contend for free medium, and lossy channel will result in corruption of transmitted packets. When either of two cases happens, corrupted packets should be re-transmitted, which increases energy consumption.
- The third source is overhearing, which happens when a node receives some packets that are destined to other nodes.
- The last one is control packet overhead. Exchanging control packets between sender and receiver also consumes some energy.

2. Scalability and self-configuration ability

As described before, for a wireless sensor network, its topology and size may change over time. So a good MAC protocol should accommodate to such changes.

3. Latency, throughput, bandwidth utilization, fairness, etc

There are common attributes for most of MAC protocols. Take latency for example, its importance depends on the actual applications. For most of sensor network applications, the speed of changes on physical objects sensed by sensor nodes is much slower than the network speed. So latency is less important and can be tolerated in a certain range in such cases.

From the above discussion, we can figure out that S-MAC, which is specially designed for wireless sensor networks, will differ from other traditional wireless MAC protocols in the following aspects: energy efficiency and self-configuration ability are the primary goals, while others attributes, like latency and fairness, are secondary. Now we introduce briefly what functions and features have been implemented in S-MAC.

The first feature that S-MAC introduces is periodic sleep and listen. The basic idea is to let each node follow a periodic sleep and listen schedule, as shown in Fig. 2.1. In listen period, the node wakes up for performing listening and communicating with other nodes. When sleep period comes, the nodes will try to sleep by turning off their radios. In this way, the time spent on idle listening can be significantly reduced, which accordingly saves a lot of energy, especially when traffic load is low. The duty cycle is defined as the ratio of listen period to a complete sleep and listen cycle. In S-MAC, the low-duty-cycle mode is the default operation for all nodes. We introduce this feature in section 2.2.

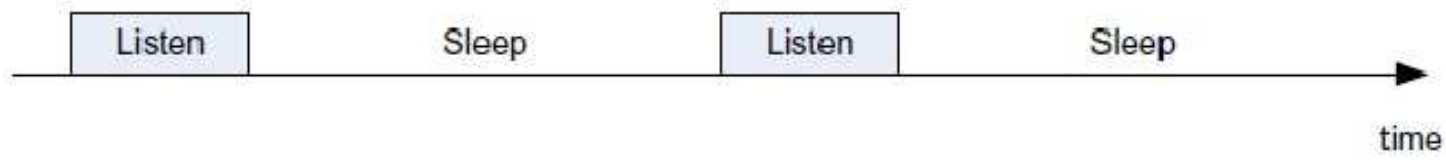

Fig. 2.1. Periodic listen and sleep

For schedule-based MAC protocols, to synchronize nodes is a serious problem. To achieve maximum energy saving and improve latency, S-MAC defines a complete synchronization mechanism, including periodic SYNC packets broadcast, schedule table and neighbor list

maintenance, periodic neighbor discovery, periodic neighbor list updating, etc. The basic idea is as follows. Each S-MAC node puts the information of its own schedule in a SYNC packet and broadcasts it to its neighbors periodically. When a new node joins the network, it will try to follow an existing schedule before choosing one by itself. When hearing different schedules on its neighbors, the node will record them in a table and follow all of them. More details about synchronization will be introduced in section 2.3.

2.2 Periodic Sleep and Listen

The main technique used to reduce energy consumption in S-MAC is to make each node in the network follow a listen and sleep cycle. A complete cycle of listen and sleep period is called a *frame*. During sleep period, the node will turn off its radio if possible. In this way, a large amount of energy consumption caused by unnecessary idle listen can be avoided especially when traffic load is light. During listen period, the node may start sending or receiving packets if necessary. S-MAC provides a controllable parameter *duty cycle*, whose value is the ratio of the listen period to the frame length. In fact, the listen period is normally fixed according to some physical and MAC layer parameters. The user can adjust the *duty cycle* value from 1% to 100% to control the length of sleep period. Normally, the frame length is the same for all nodes in network.

The listen period is further divided into two parts. The first one called SYNC period is designed for SYNC packets, which are broadcast packets and used to solve synchronization problems between neighboring nodes. We will discuss all synchronization problems in S-MAC in section 2.3. The second one called DATA period is designed for transmitting DATA packets. The format of S-MAC frame is shown is Fig. 2.2

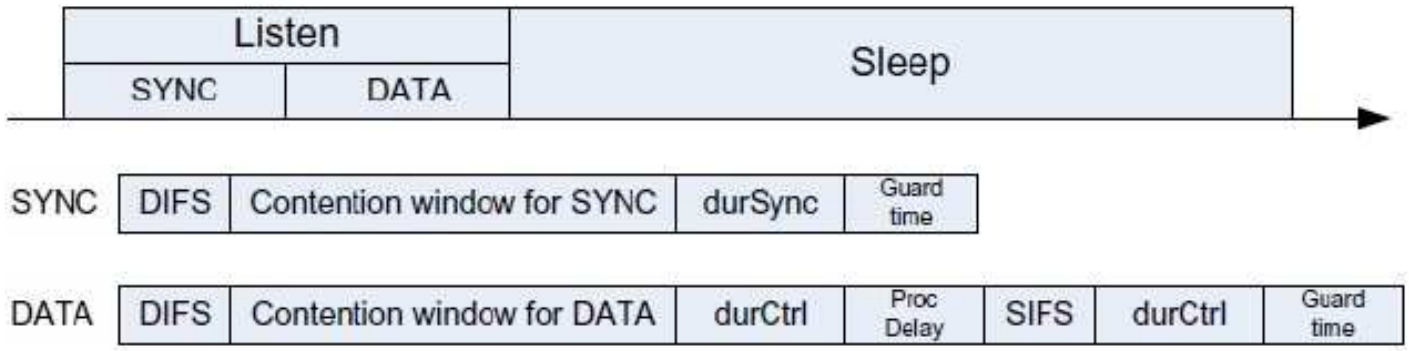

Fig. 2.2. S-MAC frame format

Each S-MAC node should have at least one schedule to follow. Each schedule is controlled by a schedule timer, which can reschedule the next period when it expires at the end of a current period. In Fig. 2.3 below, we can see that each frame will have three expiration points, which are called checking points.

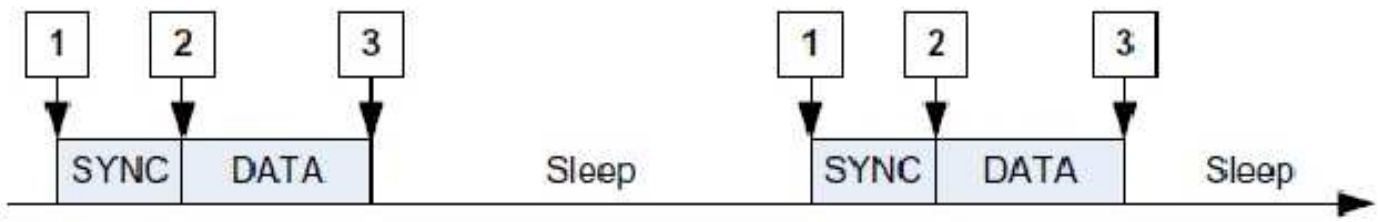

Fig. 2.3. Three checking points in a frame

At any of these checking points, S-MAC will decide what to do in the coming period. For example, at checking point 2, S-MAC checks if it has a data packet in the buffer to send. If yes, it will start carrier sensing. Otherwise, S-MAC will never try to send any data packet in this frame, if adaptive listening is not applied (adaptive listening is introduced in section 2.7). In other words, even if S-MAC receives a new data packet from its upper layer right in the middle of the DATA period, it will buffer it and wait until the next checking point 2 comes, rather than start contending for the medium right away.

Letting the node follow a sleep and listen schedule doesn't mean that the node must keep listening in the listen period or must go to sleep when the scheduled sleep period comes. The actions of S-MAC at a certain time depend not only on the current scheduled period, but also on

some other factors. Those factors include current MAC state, radio state, channel state, neighbors' state, etc.

For example, at checking point 3, S-MAC will go to sleep by turning off its radio, only when all the following listed conditions are satisfied.

- My radio is not in receiving or sending state.
- If I have not only one schedule in my schedule table (a border node), this checking point must belong to my primary schedule. (Primary schedule is introduced in section 2.3.1)
- I am idle, neither a sender nor a receiver of an ongoing data transmission. For example, I cannot go to sleep, if I have sent back a CTS packet to my neighbor in last DATA period and am now waiting for the data packet.
- I am not in a neighbor discovery period. (to be introduced in section 2.3.4)
- I am not executing adaptive listening now. (to be introduced in section 2.7)

In this section, we mainly talk about the basic idea of schedule mechanism in S-MAC. In latter sections, you will see that not only the sleep and listen schedule, but also other features of S-MAC, such as overhearing avoidance and adaptive listening, can make the node go to sleep or wake the node up.

In shared-medium networks, how to avoid collision is a common topic for all contention-based MACs. In this aspect, S-MAC is quite similar with the DCF protocol in the IEEE 802.11 standards. The features that S-MAC has adopted include physical and virtual carrier sense, RTS/CTS/DATA/ACK sequence for hidden station problem. The contents regarding this part are put in section 2.4.

Overhearing will become another major source of energy waste, especially when the node density is high and the traffic load is heavy in the network. S-MAC tries to avoid overhearing by letting all interfering nodes, which are immediate neighbors of both the sender and receiver, go to sleep after they hear an RTS or CTS packet. We talk about this problem in section 2.5.

To efficiently transmit long messages in both energy and latency respects, S-MAC supports message passing. This function divides a long message into a number of small 8

fragments and transmits them in a burst. S-MAC uses only a pair of RTS/CTS for one message passing, but requests an ACK packet for each fragment. The RTS/CTS packets reserve the medium for transmitting all fragments. That is to say, the longer the message the node has, the more time it will occupy the medium. Obviously in this way, S-MAC trades fairness on fragment level for fairness and latency on message level. But this is reasonable for wireless sensor network applications because of its features in aspects of data management. We talk about message passing in section 2.6.

Low-duty-cycle operation reduces energy consumption at the cost of increased latency. When one node receives a data packet from its previous-hop node, it cannot retransmit the packet to its next-hop node right now. It has to wait until the next listen time for the next-hop node comes. So there exists a potential delay on each hop, when a data packet is transmitted through a multi-hop network. To reduce such latency, S-MAC proposes an important mechanism called adaptive listening. The basic idea is to give all nodes, which are involved in a transmission, an additional chance for transmitting their packets at the end of the transmission. These nodes include the sender, the receiver, and all their immediate neighbors that overhear the transmission. In this way, a data packet can be retransmitted immediately after its last transmission. We discuss adaptive listening in detail in section 2.7.

2.3 Synchronization

2.3.1 Synchronization Related Components in S-MAC

- **SYNC Packet**

As mentioned before, each S-MAC node needs to exchange its schedule by periodically broadcasting a SYNC packet to its neighbors. The period of sending a SYNC packet is called the *synchronization period*. The default value in ns-2 is 10 frames (one frame = one sleep and listen cycle). Sending or receiving SYNC packets takes place in SYNC period. The definition of SYNC packet frame in ns-2 is shown in Table 2.1.

Table 2.1. The SYNC frame

Field	Comment

type	Flag indicating this is a SYNC packet
length	Fixed size with 9 Bytes
srcAddr	ID of the sender
syncNode	ID of the sender's synchronization node
sleepTime	Sender's next sleep time from now
state	Indicating if the sender has changed its schedule recently
crc	Cyclic Redundancy Check

- Schedule Table

Each S-MAC node maintains one schedule table, which stores its own schedule and the schedules of all its known neighbors. The establishment of the schedule table will be introduced in the next subsection. There are two classes of schedules in the schedule table. We call the schedule that the node itself is on *primary schedule* and other schedules in the schedule table *secondary schedules*. A node may have no secondary schedule, for example, if all its neighbors follow the same schedule. But every node should have a primary schedule. The maximum number of records in the table is limited by a user-adjustable S-MAC parameter, which defines the maximum number of different schedules. Table 2.2. shows how each field in the schedule table is defined in ns2.

Table 2.2. Field definition of schedule table

Field	Comment
txSync	Flag indicating need to send sync
txData	Flag indicating need to send data
numPeriods	Counter for sending sync periodically
numNodes	Number of nodes on this schedule
syncNode	The node who initialized this schedule
chkSched	Flag indicating need to check numNodes

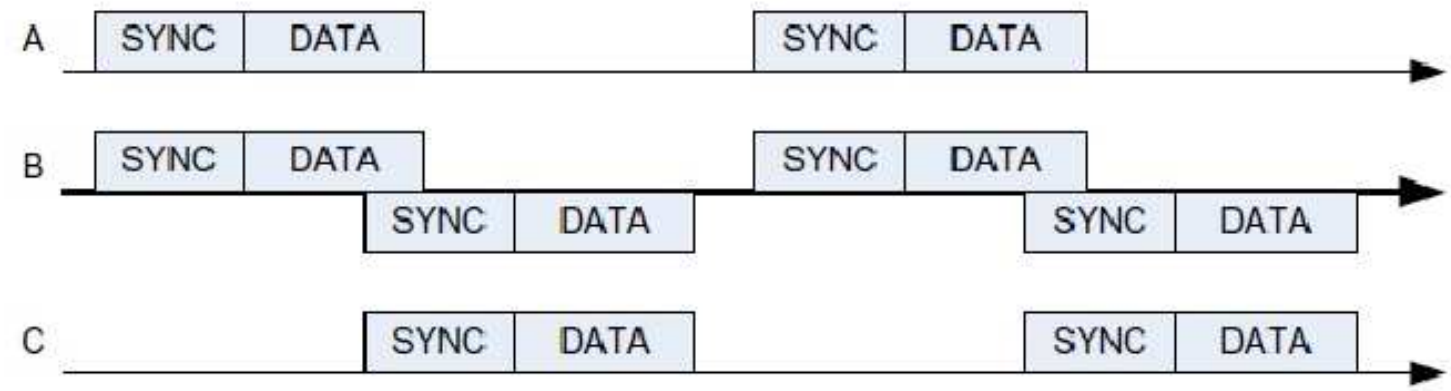

Fig. 2.4. A border node with two schedules

Now we explain why S-MAC introduces such a table. We intuitively hope that only one schedule exists in the whole network, because sleep durations and energy saving will be maximized when all nodes are on the same schedule. And we know that some neighboring nodes can get synchronized through exchanging SYNC packet between them. But in practice this mechanism takes effect only locally. The existence of multiple schedules in a network is quite common, especially in a large multi-hop network where nodes do not start initializing at the same time and join in a random way. We call those nodes that share the same schedule a *virtual cluster*, and nodes with neighbors in two or more clusters *border nodes*. Obviously, if the border node wants to talk to its neighbors at their scheduled listen time, it has to know when its neighbors wake up and go to sleep. Schedule table is designed for this purpose.

A schedule is actually a timer. Both primary schedule and other secondary schedules have their own timers. These timers run independently and their workings are quite similar. We take a simple example to illustrate how a border node works under multiple schedules. We assume that node B has two neighbors node A and C. As shown in Fig. 2.4, B has two schedules in its schedule table. One is its own schedule (primary schedule), which is the same as A's, and the other is heard form C. B will allocate a schedule timer for either of the two schedules. The two schedule timers run independently.

We first see how B broadcasts SYNC packets. To ensure that both A and C can receive the SYNC packets in their own SYNC period, B has to broadcast its SYNC packets periodically on both schedules. Please note that, in a SYNC packet, a border node always tells its neighbors the

time from now to its next sleep time according to its primary schedule, but not to any of its secondary schedule.

The similar way is applied to broadcast data packets. When B has a data packet to broadcast, it has to broadcast it twice. One takes place on A's DATA period and the other on C's DATA period.

Now we discuss unicast data packets. Suppose B receives a data packet destined to C from the upper layer, it first searches in its schedule table to find out the schedule that C is following, and it sets the txData flag on that schedule to 1. When C's next DATA period comes, the corresponding schedule timer on B will inform B that its data buffer has a data packet destined to C and now it is time to send it out.

From the above discussion, we can see that a border node has to follow several schedules to get synchronized with its neighbors that are on the different schedules. Obviously, border nodes will consume much more energy than non-border nodes. The paper [3] proposes an algorithm called *global schedule algorithm* that allows all nodes in a multi-hop network to converge on a single global schedule. This algorithm has been implemented in S-MAC in ns-2.28. However, studying S-MAC under multiple schedules is not the intention of this thesis.

- **Neighbor List**

Another important component in S-MAC is *neighbor list*. Each S-MAC node has to set up such a table to records the information of all its known neighbors. The number of records in the list is also limited by a user-adjustable S-MAC parameter, which defines the maximum number of neighbors for each node. Like schedule table, neighbor list is established also through exchanging SYNC packets between neighboring nodes. Table 2.3 below shows the definition for each field in neighbor list.

Table 2.3. Field definition of neighbor list

Field	Comment
nodeId	ID of this node
schedId	Schedule ID in schedule table that this node follows
active	Flag indicating this node is active recently
state	Flag indicating the node has changed schedule

Neighbor list plays an important role in S-MAC. When S-MAC processes a unicast data request received from the upper layer, it will first check its neighbor list to see if the destination node is on the list. If not, the request will be refused. If yes, the flag txData of the schedule that the destination node follows is set (if no other sending request). Then when the next DATA period on this schedule arrives, the node will try to send out this packet.

2.3.2 Choosing First Schedule

When a new node joins the network, it first listens for a fixed period (normally a synchronization period). We call this period *initial listening*. Either of the following two cases may happen.

1. No SYNC packet received during the initial listening.

At end of this period, the node chooses a schedule by itself and sets a schedule timer for it. Normally the first cycle starts with SYNC period. Meanwhile the schedule is added to the first entry of its schedule table. To announce this new schedule, the flag txSync of this schedule is set, which indicates that the node will try to broadcast a SYNC packet in the next SYNC period.

2. Get a SYNC packet before the initial listening ends.

That is the node's first SYNC packet received. It immediately follows the schedule that comes with the received SYNC packet, instead of choosing by itself after the initial listening. The value of sleepTime in the SYNC packet determines the starting point of the first cycle. Meanwhile the schedule is added to the first entry of the schedule table, and the sender of the SYNC packet is added to the neighbor list. The node also needs announce this schedule by setting the flag *txSnyc* in this schedule.

2.3.3 Updating and Maintaining Schedules

Now we know from the last subsection that after initial listening the node may get its first schedule by choosing itself or by following an existing one. In this subsection, we discuss how S-MAC maintains its schedule table and neighbor list every time it receives a SYNC packet from one of its neighbors after it has chosen its first schedule.

1. **This is my first SYNC packet.**

This may happen only after I have chosen my first schedule by myself (option 1 in last sub-section). In this case, I will discard my first schedule by removing it from my schedule table and follow the new one in the SYNC packet by adding it to the first entry of the schedule table. And the schedule timer that is associated with the discarded schedule will be rescheduled according to the value of sleepTime in the SYNC packet. Of course, the node that sent this SYNC packet will be added to my neighbor list (my first neighbor).

2. This is not my first SYNC packet after I have chosen my first schedule.

We have to consider the following five possible situations. For simplification, we use N representing the sender of this SYNC packet and S representing the schedule in the SYNC packet. The algorithm is put in Table 2.4 below.

Table 2.4. General algorithm for processing SYNC packets

	Condition	Action
1	N is a known neighbor in my neighbor list and it has not changed its primary schedule since I got its last SYNC packet.	Update S in my schedule table by rescheduling its schedule timer with the value of sleepTime in the SYNC packet. This can eliminate clock drift between two nodes.

2	N is a known neighbor in my neighbor list, but it has switched its primary schedule to S, which is new to me, since I got its last SYNC packet.	Step1: Process the schedule that N has switched from. The number of the nodes on this schedule has to be decreased by one. If the number goes to 0 after decrease, this schedule has to be removed from my schedule list. If now I happen to have a DATA sending request, I must defer changing till the sending is over. Step2: Process the new schedule S that N has switched to. If my schedule table is not full, S is added and assigned to a new schedule timer. Otherwise N has to be deleted from my neighbor list. Step3: If the schedule that N switched from is my primary schedule, I need to execute *check_my_schedule*, which checks if I become the only one on my primary schedule. If yes, I have to choose the next available schedule in my schedule table and set it as my new primary schedule (delete the old one). But if now I happen to have a DATA sending request, I must defer *check_my_schedule* until the current sending is over.
3	N is a known neighbor in my neighbor list, but it has switched its primary schedule to S, which is an existing one in my schedule table, since I got its last SYNC packet.	Step1: Process the schedule that N switched from. The number of the nodes on the schedule that N used to follow has to be decreased by one. If the number goes to 0 after decrease, this schedule has to be removed from my schedule list. If now I happen to have a DATA sending request, I must defer changing till the sending is over. Step2: Process the schedule S that N has switched to. Update S in my schedule table by rescheduling its schedule timer with the value of *sleepTime* in the SYNC packet. And the number of nodes on S is increased by one.
4	N is an unknown neighbor to me and S is also new to me.	If neither my schedule table nor neighbor list is full, add S to my schedule table and assign a new schedule timer to S. Then add N to my neighbor list.
5	N is an unknown neighbor to me, but S is known in my schedule table.	Update S in my schedule table by rescheduling its schedule timer with the value of sleepTime in the SYNC packet. If my neighbor list is not full, N is added to it. And the number of nodes on S is increased by one.

2.3.4 Periodical Neighbor Discovery

In S-MAC, neighboring nodes discover each other through exchanging SYNC packets. However, sometimes two neighboring nodes may miss each other forever, for example, when they follow different schedules, whose SYNC periods do not overlap. Periodical neighbor discovery can prevent this from happening.

The basic idea of neighbor discovery in S-MAC is to make each node periodically execute neighbor discovery for a whole synchronization time. During neighbor discovery time, S-MAC will never try to go to sleep when its sleep period comes, so that the node can listen more time than usually and have more chances to hear a new neighbor. For a border node, neighbor discovery is only executed on its primary schedule. The reason is that on secondary schedule, the node will not try to go to sleep when scheduled sleep period arrives.

Fig. 2.5 shows how the neighbor discovery period is defined and its relationship with other periods defined in S-MAC. The neighbor discovery period may vary, depending on the current number of its known neighbors. In ns-2, the period of executing neighbor discovery is 2 synchronization periods if the node has no neighbor. Otherwise, the period is much longer and reaches 33 synchronization periods.

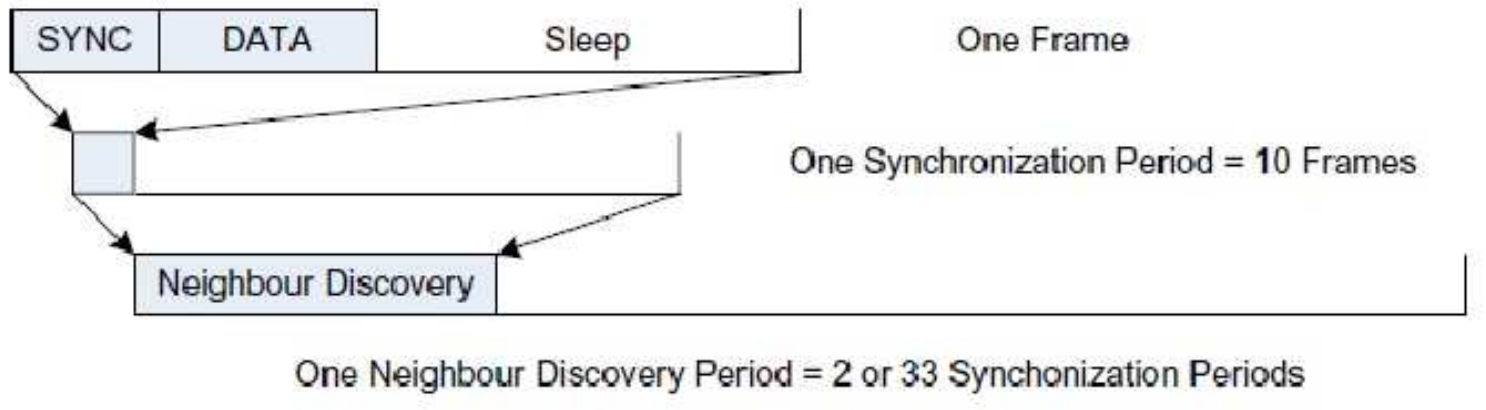

Fig. 2.5. Hierarchy of periods in S-MAC

2.3.5 Periodical Neighbor List Updating

The schedule table and neighbor list will be updated each time when the node gets a SYNC packet from one of its neighbors. Except that, each S-MAC node also needs to check its neighbor list periodically to see if some of its neighbors have been inactive for a certain time (moved away or died for some reason), which need to be removed from the neighbor list. Doing this is necessary and important, because it cannot only save space for the neighbor list, but also avoid unnecessary energy consumption caused by keeping trying to talk to an inactive neighbor, especially in a multi-hop mobile sensor network.

A timer is allocated to control periodical updating of the neighbor list. Its expiration period must be larger than the synchronization period. The reason for it will be explained later. When the timer expires, the following steps will be executed. (Relevant methods in the source file smac.cc in ns-2.34 includes handleUpdateNeighbTimer(), update_myNeighbList(), update_neighbList())
Step 1: Check if I have a DATA sending request now. If yes, skip the following steps and defer updating till the current sending is over. If no, from now, I will be temporarily disabled from receiving new requests from the upper layer to avoid error operation during updating. (In ns-2, this is done by setting the flag txRequest to 1, which prevents S-MAC from receiving a new request from the upper layer.) .

Step 2: Update the number of nodes on a certain schedule if its flag *chkSched* is set. This may happen when I got a SYNC packet from one of my known neighbors, which had switched to another schedule since I got its last SYNC packet, and I should decrease the number of nodes on the schedule that this neighbor had switched from. But at that time I had a DATA sending request, so I had to defer decreasing. And now is time to do it.

Step 3: Update neighbor list. Check each record in my neighbor list to see whether its *active* flag is set or not.

- If yes, the flag is reset. (The flag will probably be set again when I get this neighbor's SYNC packet next time before my next time for neighbor list updating arrives.)
- If no, which means that I have not received the SYNC packet from this neighbor node for a long time (at least a neighbor list updating period), this neighbor node has to be deleted from my neighbor list. Accordingly, the number of nodes on the schedule that the deleted neighbor used to follow should be decreased. If after decrease the number becomes 0, this schedule has to be removed from my schedule table.

Step 4: If the inactive neighbor nodes that I dropped in step 3 are on my primary schedule, I have to execute *check_my_schedule*, which has been introduced in Table 2.4 Part 2.

Step 5: Reset the timer to schedule next neighbor list updating and enable receiving new requests from the upper layer again.

Now we explain the reason why the period for updating neighbor list must be longer than the period for synchronization. Fig. 2.6 shows an example, which has two neighboring nodes A and B. We assume the period for updating neighbor list is smaller than synchronization period, and A can always receive SYNC packets from B successfully. We can see from the figure that A executes neighboring list updating twice in one synchronization period for B. It is easy to imagine that, at the second updating time for A (red arrow), A thinks that B has not been active recently and removes B from its neighbor list, because A has not got B's SYNC packet since last time when it updated its neighbor list. B will become unknown to A until B's next broadcasting time comes. In this blank period, A cannot talk to B, because it takes it for granted that B is not active. However, this is not true, because B keeps active all the time.

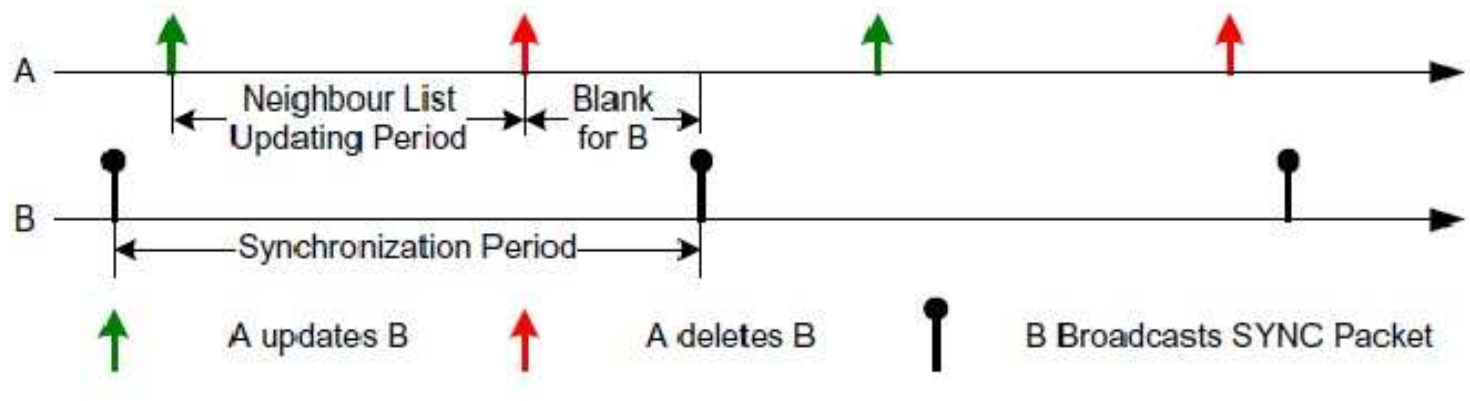

Fig. 2.6. Neighbor list updating period smaller than synchronization period

2.4 CSMA/CA in S-MAC

S-MAC achieves energy saving mainly by utilizing the schedule mechanism. The schedule controls nodes' periodic sleep and listen. From this point of view, S-MAC looks like a schedule-based MAC protocol, e.g. TDMA. However, with respect to transmission control and collision avoidance, S-MAC is more like a contention-based MAC protocol, especially like the DCF (Distributed Coordination Function) in IEEE 802.11 protocols. In this section, we will introduce the CSMA/CA mechanism in S-MAC, including carrier sense (including physical and virtual mechanisms) and RTS/CTS mechanism for collision avoidance and hidden station problem.

2.4.1 Carrier Sense

Like DCF in 802.11, carrier sense in S-MAC is performed both through physical and virtual mechanisms. The medium is determined as free only when both of them indicate that it is free.

Physical carrier sense is performed at physical layer by checking the current radio state. Every time when the radio starts receiving or transmitting, the PHY layer will inform the MAC thereof. It happens also when the receiving or transmitting is overObviously, the medium will be determined as busy when radio is in receiving or transmitting state.

Virtual carrier sense is performed at the MAC layer. We know that in 802.11, each station maintains a so-called network allocation vector (NAV), which is actually a counter. It will count down to zero at a uniform rate after it is assigned with a certain value. NAV mechanism requests that each packet sent by MAC contains a duration field, which indicates how long the current transmission will last. When a node receives a packet destined to another node, the duration value in the packet will tell the node how long it has to keep silent. The node's NAV will be updated with the duration value, only if the duration value is larger than the current NAV value. Virtual carrier sense indicates that the medium is idle, when the NAV value is zero. When non-zero, the indication is busy.

The similar virtual carrier sense mechanism is also applies to S-MAC. However, unlike that in 802.11, S-MAC defines two NAVs. One is simply called *NAV* and used to indicate if the medium is busy or not, just the same as in 802.11. The other one is called *Neighbor NAV*, which tracks its neighbors' NAV. Only when both of the NAVs have counted down to zero, the virtual carrier sense indicates the medium is free. The structure of and the way of running for two NAVs are quite similar. Table 2.5 lists all events that make two NAVs update their values. If not mentioned, the duration value used to update the NAV value is always the duration value in the packet that is received or sent in corresponding event.

Table 2.5. Differences between NAV and Neighbor NAV

NAV	Neighbor NAV

Receive RTS destined to other node	Receive RTS destined to me
Receive CTS destined to other node	After CTS is sent out (for DATA timeout)
Receive DATA destined to other node	Receive DATA destined to me
Receive ACK destined to other node	After DATA (unicast) is sent out
Error found in received packet, NAV is updated by an EIFS value	After ACK is sent out
Collision detected when the radio is receiving a packet and another packet arrives, NAV will be updated by an EIFS value right after the radio finishes receiving	ACK timeout, but not reached maximum number of times to extend Tx time

From the above table, we can see that one node uses its NAV for virtual carrier sense when it interferes in one of its neighbor's transmission, and uses its Neighbor NAV when it is either a sender or a receiver during a transmission process. Besides their functions for virtual carrier sense, the NAVs also play other roles in S-MAC listed as below.

1. Neighbor NAV will act as a timer for DATA timeout on the node, which has sent out the CTS packet and is waiting for the arrival of the DATA packet.

2. Adaptive listening (to be introduced in latter parts of this chapter) will be triggered when either of the two NAVs counts down to zero.

S-MAC obtains a transmission chance through contending for the medium in a contention window. Going back to Fig. 2.2, we find that there are two contention windows defined in one frame. One is for sending SYNC packets in the SYNC period and the other is for sending DATA packets in the DATA period. Both contention windows have fixed size (fixed number of slots), which must be 2n-1, e.g. 31 slots for SYNC and 63 for DATA. Actually, S-MAC can have a third contention window in the sleep period, if adaptive listening is applied (We will discuss adaptive listening in section 2.7). Here we consider only S-MAC without adaptive listening. Now we take a simple example and see how carrier sense is performed before sending a SYNC

packet in the SYNC period. The following steps are also applicable to sending a DATA packet in the DATA period.

When the schedule timer expires at the end of the sleep period indicating the coming of a new SYNC period, it will check the conditions for sending a SYNC packet. If the syncFlag on this schedule is set and both of the NAVs have zero values (virtual carrier sense indicates free medium), the node will be ready to broadcast a SYNC packet. If the radio is in sleep state, it will be awoken first. Then the node starts physical carrier sense. The duration for carrier sense is chosen uniformly within the contention window and consists of a DIFS. One of two following possibilities may happen during this period.

- If nothing is heard throughout the whole carrier sense period, the node will assume that the medium is free and start transmitting the packet right away.
- Once the medium is sensed busy during the carrier sense period, the node will stop sensing and defer sending the packet. When the same period in the next frame arrives, the node will retry sending. While retrying, it will follow the same procedures described above.

In the above discussion, S-MAC always chooses the number of slots for carrier sense within the fixed contention window size, uniformly and independently. This is not like the binary exponential back off introduced in 802.11.

2.4.2 Collision Avoidance

We know that the RTS/CTS mechanism defined in the DCF can efficiently reduce the durations of collisions and solve the so-called hidden station problem. This medium reservation technique has been introduced into S-MAC. Prior to the actual DATA packet, the sender should exchange RTS and CTS packets with the receiver. But only unicast packets follow the sequence of RTS/CTS/DATA/ACK. Broadcast packets will be sent without using RTS/CTS. In this sub-section, we discuss unicast packet only.

We have learnt that the virtual carrier sense mechanism introduced in subsection 4.2.1 is achieved by distributing reservation information announcing the impending use of the medium. And the reservation information is recorded in the duration field of RTS/CTS/DATA/ACK packets. RTS/CTS packets will reserve for the whole transmission process. All immediate neighbors of both the sender and the receiver will learn the coming transmission from the RTS or CTS packets and will keep silent during the reserved period. Although the combination of carrier sense and RTS/CTS mechanisms can largely reduce the probability and durations of collisions, collisions cannot be completely avoided. If two neighboring nodes finish carrier sense and start sending at almost the same time, collision will occur. We take a simple example to see how S-MAC sends a data packet out and assume that the channel is ideal (no corruption case).

Consider two neighboring nodes A and B, which follow the same schedule and have discovered each other. A receives a sending request destined to B from the upper layer. We also assume that A does not have any other sending request at present. The carrier sense does not start immediately and needs to wait until the next DATA period comes. A will follow the steps described in last sub-section to start carrier sense.

Fig. 2.7 below shows one possible case. We can see from the figure that A and B exchange RTS/CTS within the DATA period and use their scheduled sleep time for the data packet transmission. Actually the length of DATA period is carefully designed. It is fixed according to some physical-layer and MAC layer parameters, e.g. the radio bandwidth and the contention window size. S-MAC has one feature in the DATA period that no matter how many slots for carrier sense have been chosen from the fixed size contention window, the exchange of RTS and CTS can always be done within the DATA period if no collision and corruption occurs. But the transmission of the DATA packet normally needs to be extended to the schedule sleep period. Normally, S-MAC is working under a relatively low duty cycle (e.g. 10%). The sleep period in a frame is much longer than the DATA period. So in normal cases, the whole transmission process can be done in one frame and will not be prolonged to the next SYNC period. In the following example, the transmission between A and B ends before the next SYNC period. If adaptive listening is not applied, both A and B will still have timeto sleep.

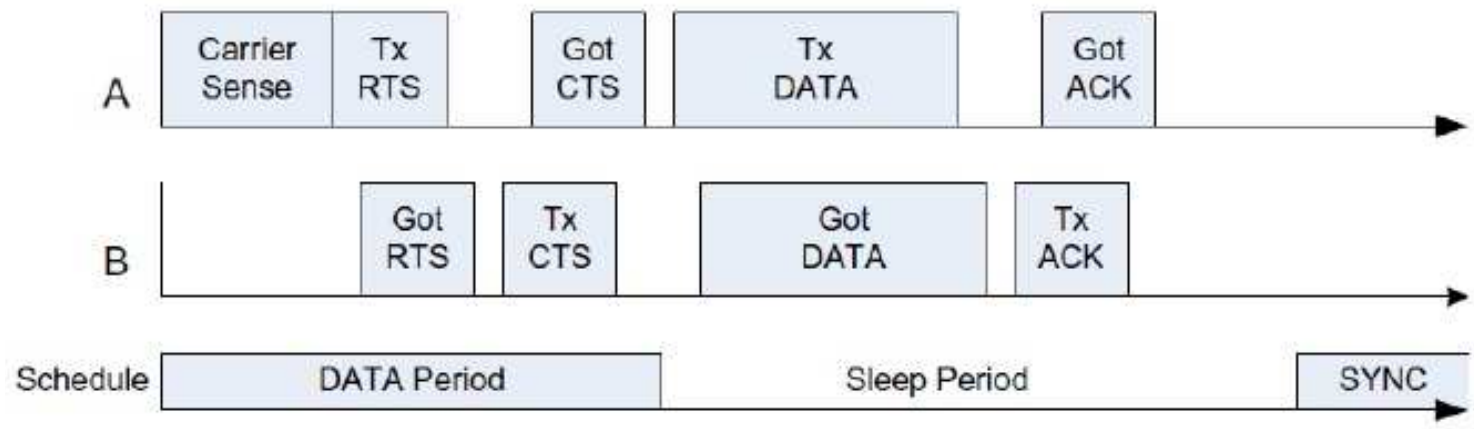

Fig. 2.7. A successful data transmission between two nodes

Now we see what will happen when collision occurs. If B also sent out the RTS at almost the same time when A sent the RTS out, two RTS packets collide. Neither of them will receive the RTS packet from the other and send a CTS packet back. After a short wile, A's CTS timer expires. If now A's primary schedule is in the sleep period, A will go to sleep by turning its radio off. When the next DATA period on the same schedule comes, A will resend the RTS. The maximum number of RTS retries for sending a DATA packet is user-adjustable in S-MAC. When the retry times reach the maximum number, A will give up sending the DATA packet and signal its upper layer about the failure of sending. The same procedures will be followed when B's CTS timeout timer expires.

2.5 Overhearing Avoidance

For contention-based protocols like IEEE-802.11, overhearing is one of the major sources of energy waste. Overhearing takes place on a node when it receives some packets that are destined for other nodes. In 802.11, measures like latency and bandwidth utilization are considered in the first place. To achieve better performance in a shared-medium network, carrier sense, especially virtual carrier sense should be performed more efficiently. The best way to achieve it is to let each node keep listening to all its neighbors' transmissions. Obviously, this method will lead to large amounts of energy consumptions on overhearing, especially when node density is high and the traffic load in the network is heavy.

For S-MAC, saving energy is its primary goal. To avoid overhearing, S-MAC forces interfering nodes to go to sleep after they receive an RTS or a CTS packet that is not destined for them. In this way, nodes that interfere their neighbors' transmissions will not hear DATA packets, which normally take much longer transmission time than control packets, and following ACK packets. We take an example in Fig. 2.8 to illustrate this algorithm.

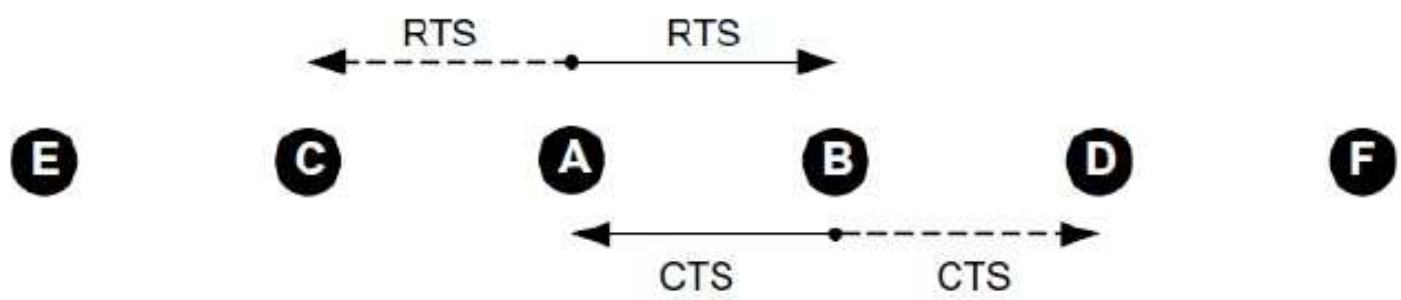

Fig. 2.8. C and D overhear the transmission between A and B

The above figure shows a five-hop linear network. Each node can only hear its immediate neighbors. We assume that all the nodes share the same schedule. Suppose A is communicating with B in the same way that we discussed in Fig. 2.7. We first see which nodes should go to sleep during this transmission.

We know that collision only happens on the receiver side. Obviously, D is supposed to go to sleep, because its transmission interferes with B's reception of the DATA packet. C is two hops away from B, so C's transmission will not interfere with B's reception. But if C talks to E while A is sending data to B, C will not receive any packet from E because collision happens on C. C's transmission is a waste of energy and it also needs to go to sleep. Both E and F are at least two hops away from the nodes that are transmitting, and they will never produce interference. Therefore, E and F have no need to sleep. In a word, those nodes that are the immediate neighbors of the sender or the receiver should go to sleep.

Now we discuss when C and D should go to sleep. We first see C, who is A's immediate neighbor. C can hear the RTS packet that A sends to B. However, C does not go to sleep after receiving the RTS, because at this moment the communication has not been really set up and may be cancelled for some reason, e.g. collision on RTS packets. Therefore, it has to keep listening for a CTS timeout period to see if it can hear the following CTS packet. If C receives

the CTS packet, it will go to sleep right now. However, in this example, C cannot hear the CTS that B sends back to A. Therefore, after the CTS timeout period, C will keep listening for another CTS timeout period and try to hear the DATA packet. As soon as C hears the DATA packet, it knows from the duration field in the DATA packet that how long exactly the current transmission will last and goes to sleep right now. If no DATA packet is heard in this period, C will go to sleep only when C's primary schedule is now in the sleep period. For D, who is B's immediate neighbor, it can only hear the CTS packets sent by B. D will go to sleep right after it gets the CTS and updates its NAV with the duration value in the CTS and goes to sleep right after receiving the CTS.

2.6 Message Passing

Except that overhearing leads to energy waste in contention-based protocols like IEEE 802.11, control packet overhead will be another reason. Control overhead means cost (energy or time) spent on exchanging control packets (RTS/CTS/ACK) when unicasting a data packet. In this section, we talk about how S-MAC reduces energy consumptions caused by control overhead.

We know that transmitting a long message using a single data packet through a lossy channel is hazardous and riskful. Even when a few bits in the packet are corrupted during the transmission, the whole packet must be re-transmitted. This will waste a lot of time and energy. Therefore, some MAC protocols like 802.11 support a fragmentation mechanism, which breaks a long message into some small fragments. All fragments are sent in a burst, and using one pair of RTS/CTS, if none of them is corrupted. If one of the fragments is corrupted, another pair of RTS/CTS is needed. When receiver has gotten all fragments, its MAC is responsible for assembling all the fragments into a whole and passing it upwards.

S-MAC adopts a modified fragmentation mechanism for transmitting a long message, called message passing. Its basic idea is to fragment a long message into many small fragments and send them in a burst. This means only one pair of RTS/CTS is used for all fragments, but the receiver should acknowledge each fragment it receives. RTS and CTS reserve the medium for transmitting all the fragments. When one of the fragments is corrupted during transmission, ACK timeout will happen on the sender side. The sender has to extend the reserved transmission time

for one more DATA/ACK pair and resend the lost fragment immediately. S-MAC sets a limit on how many extensions can be made for one message. This prevents some nodes from occupying the medium for too long in some special cases, e.g. the receiver loses its connection with the sender during the transmission.

As for neighboring nodes that may interfere with the ongoing transmission, they will follow the same way described in section to go to sleep after they hear the RTS or CTS. However, some special cases may happen. For example, the neighboring nodes miss the chance of receiving the RTS or CTS for some reason, e.g. they were sleeping at that time, and they wake up afterwards. Or a new node happens to join the network in the middle of the transmission. Then what should these nodes do in these cases? Actually in S-MAC, besides RTS and CTS, both DATA fragment and ACK packets have the duration field, which is now the time needed for transmitting all the remaining data fragments and ACK packets. Fig. 2.9 illustrates how to set duration value (NAV value) for each packet in the transmission of a 3-fragment message in a burst. So no matter which of them is received by the interfering neighbor nodes, they will know the time the current transmission will last and how long they should sleep. This is also the reason why S-MAC requests the receiver to acknowledge each DATA fragment that it receives. Using the Fragment/ACK pair in message passing resembles the RTS/CTS pair in the way it solves the hidden station problem. Sending the ACK frequently will make the neighboring nodes, which can only hear the receiver, update their NAVs in time. For example, again in Fig. 2.8, D goes to sleep right after it has received the CTS from B. However, one fragment is corrupted during the transmission and the original reserved time for the whole transmission has to be extended. When the original reserved time is over, D may wake up from sleep. It will know the extended transmission time from the ACK packet and go to sleep again.

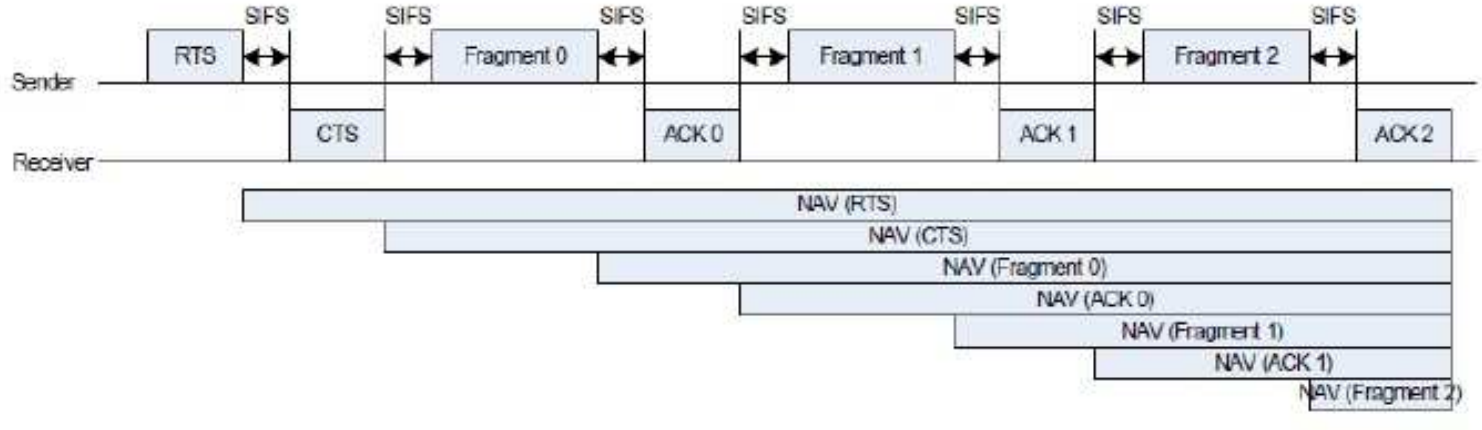

Fig. 2.9. RTS/CTS/DATA/ACK and NAV settings in message passing

From the above discussion, we find that the message passing mechanism in S-MAC can effectively reduce the control overhead in transmitting a long message, by sending all fragments in a burst and using only a pair of RTS/CTS for it. When the sender fails to get an ACK for any fragment, message passing will provide some opportunities for re-transmitting the lost or corrupted fragment, instead of giving up the transmission immediately and re-contending for the medium. This approach seems to be unfair to those nodes that have a short message to send. However, in this way, application-level performance for each node can be improved greatly, if compared to the traditional MAC protocols. That is what sensor networks desire.

2.7 Adaptive listening

In this section, we introduce adaptive listening, which is one of the most important and attractive features in S-MAC. In the previous sections, we have mainly discussed how S-MAC reduces energy consumptions. Among those techniques, the scheme of periodic listen and sleep contributes the most. On the other hand, the schedule mechanism will increase the latency of sending packets in a multi-hop network. Although both synchronization mechanism and message passing mechanism have decreased the latency to a certain degree, adaptive listening is the dominant technique that S-MAC has proposed to efficiently and largely reduce the latency in a multi-hop transmission.

Let us look at an example to see how S-MAC transmits a packet in a three-hop linear network. Suppose there are four nodes A, B, C, and D, which are put in a line, as shown in Fig 2.10. Each node can hear only its immediate neighbors. A is the source and D is the sink. Now a data packet is generated at the source node and destined to the sink node. We assume all nodes share the same schedule and no message passing technique is applied.

Fig. 2.10. A three-hop network with one source and one sink

We first see the case shown in Fig. 2.11. When adaptive listening is not employed, each node has at most one chance to send the data packet out in each frame, because checking to send the data packet only takes place at the beginning of each DATA period. If the arrival of the data packet at a certain node misses this checkpoint of this frame, it has to wait until the next DATA period comes. That is to say, when each node strictly follows its sleep schedule, there is a potential delay on each hop. We can see from the Fig. 2.11 that we need three frames to transmit the data packet from the source A to the sink D. The data packet travels only one hop in each frame.

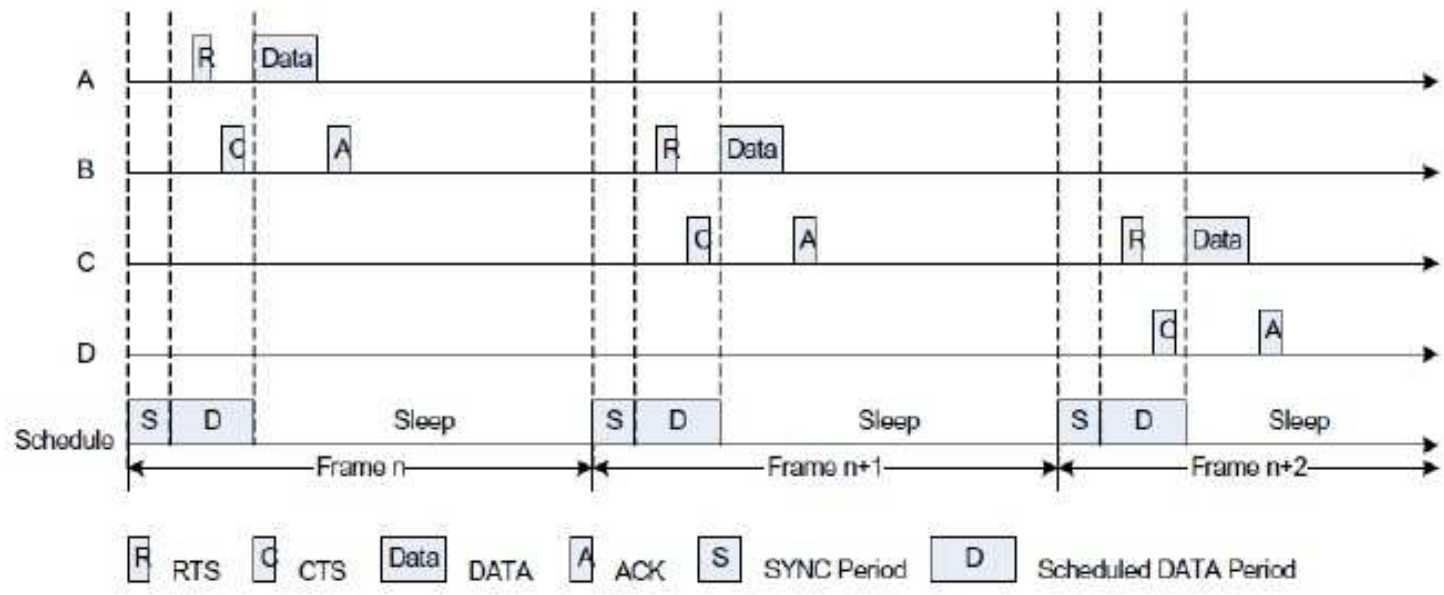

Fig. 2.11. Transmitting a data packet without adaptive listening

Now we consider the case if adaptive listening is applied. The basic idea is that at the end of one transmission, S-MAC will give those nodes, which are involved in this transmission including sender, receiver, and neighbors of both them that overhear this transmission, another chance of transmitting data packets. In this way, one node may27 have two DATA periods for sending or receiving data packets in a frame. We use Fig. 2.12 below to illustrate how adaptive listening functions in S-MAC.

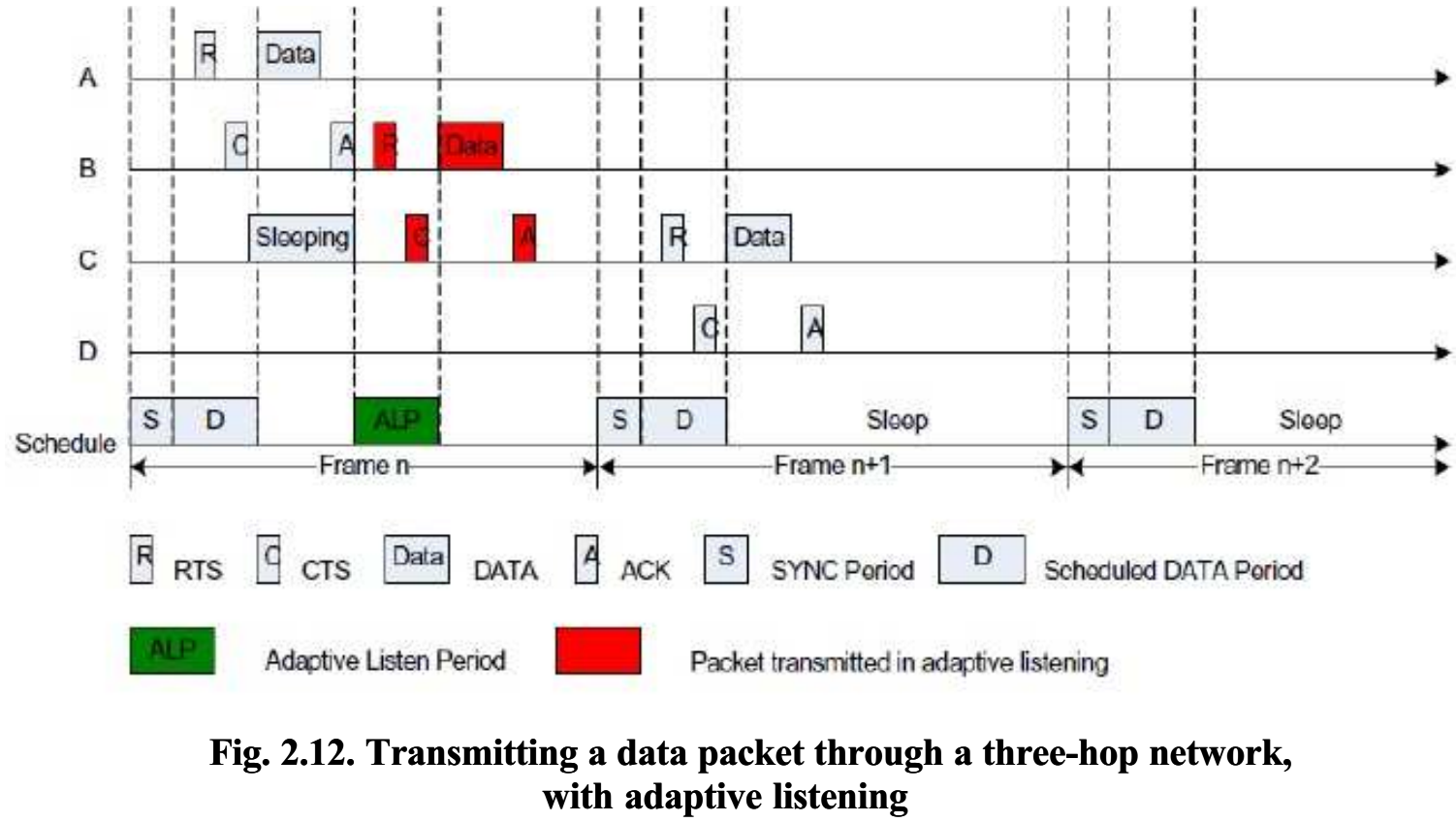

Fig. 2.12. Transmitting a data packet through a three-hop network, with adaptive listening

We start with the transmission between A and B. As shown in Fig. 2.12, transmitting the data packet from A to B (including carrier sense and exchange of RTS/CTS) starts from the scheduled DATA period and extends to the following sleep period. We have known from the previous sub-section 2.4.1 that both the sender A and the receiver B will use their Neighbor NAVs to perform virtual carrier sense, while C is overhearing this transmission and will use its NAV for virtual carrier sense. Both NAV and Neighbor NAV contain the same value, which is the time reserved for the current transmission. The expiration event of those NAVs (count down to zero, indicating the end of the current transmission) will trigger the execution of adaptive listening on their owner nodes. The following universal steps will be executed when adaptive listening is triggered at a certain node.

1). Check if the remaining time in current frame (from now to the next listen time) is shorter than a DATA period. If yes, give up executing the following steps and exit. Otherwise, go step 2.
The reason for executing this step will be explained later in this section. And we assume the example in Fig. 2.12 satisfies this condition.
2). Set a timer (called adaptive listening timer) to bring me back to sleep (if conditions allow me to sleep at that time). The expiration time for the timer is equal to a DATA period (here called adaptive listening period, see the green box in Fig. 2.12).

3). If I am still sleeping, I have to wake up now. This may happen to the node, such as C in Fig. 2.12, which is sleeping because it overheard its neighbor's previous transmission.

4). Check if the flag txData on my primary schedule is set (that is to see if I have a buffered unicast data packet to send on my primary schedule). If yes, I will try to send out the data packet by following the same way I do in the schedule DATA period (described in section 2.4). Otherwise I have to keep awake, because my neighbors, who are also adaptive listening now, may want to talk to me. Please notice that broadcast data packet will never be sent when I am adaptive listening, because I am not sure whether all my neighbors are also adaptive listening now. Some of my neighbors may not be aware of the previous transmission that I was involved in, so they do not execute adaptive listening as me and may be asleep now. For the example shown in Fig. 2.12, only B satisfies the sending condition, because it just got the data packet from A that needs to be retransmitted to C. After exchanging RTS/CTS with C successfully, B transmits the data packet to C, as shown in the red box in Fig. 2.12. For A, it has nothing to send and has to keep awake. But it will hear the RTS that B sends to C. So A has to go to sleep to avoid overhearing the transmission between B and C.
5). The adaptive listening timer set in step 2 expires. If I am still awake only because I have nothing to send and hear nothing during this adaptive listening period, now I will go back to sleep again. For the example in Fig. 2.12, no node goes to sleep for such reasons stated above. Both B and C are impossible, because B is transmitting to C. And A overheard this transmission

and may have gone to sleep before its adaptive listening timer expires. Now we explain why S-MAC checks the remaining time in the current frame before starting adaptive listening. From the above discussions, we see that the essential of adaptive listening is to add another DATA period (adaptive listening period) in the scheduled sleep period, so that nodes can obtain an additional chance for sending or receiving data packet. And we do not want to see the adaptive listening period overlap the next coming SYNC period, because this may interfere with the transmission of the SYNC packets. So that is the reason. But we have to realize one fact, that is, even if the adaptive listening period is located exactly inside the sleep period, the transmission may extend to the next SYNC period. One example for such cases is shown in Fig. 2.13. So we can say only that making the adaptive listening period not overlap the next SYNC period is to give the priority to the SYNC packets, but adaptive listening does not assure that it never collides with the next SYNC period.

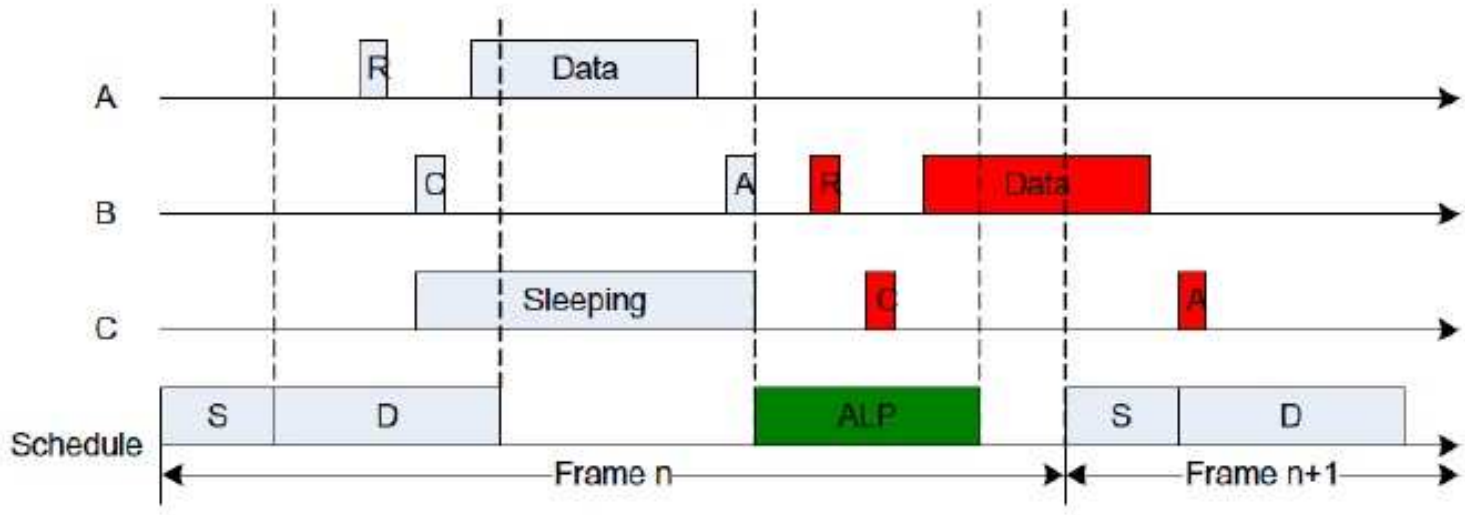

Fig. 2.13. Adaptive listening extends to the next frame

From the above discussion, we see that adaptive listening can greatly reduce the latency caused by the periodic sleep of each node in multi-hop networks. Consider one data packet, which is to be transmitted through a multi-hop network. When adaptive listening is not applied, the data packet can jump only one hop in a frame time. When adaptive listening is applied, the data packet can be retransmitted to the next-hop node immediately after its last transmission is over.

It should be mentioned that theoretically the range of adaptive listening is limited to one hop, because we assume that neighbors that are two hops away cannot hear each other. Under this

assumption, one data packet is able to jump at most two hops in a frame time. We use Fig. 2.14 to illustrate this problem. At the end of the transmission between A to B, the adaptive listening is triggered and B starts to retransmit the data packet to C. As happened before, at the end of the transmission between B and C, a second adaptive listening will be triggered. We assume that the remaining time from now to the next listen time is long enough to accommodate an adaptive listening period. Now C tries to retransmit the data packet to D and sends out a RTS packet after carrier sensing. But obviously D is sleeping, because D has not been aware of the adaptive listening that has happened to its neighbors. So C will encounter a CTS timeout and have to wait until the next DATA period comes.

Fig. 2.14. Adaptive listening triggered twice in a frame

Chapter 3

Simulation using Network Simulator-2 (NS-2)

3.1 Introduction

Network Simulator, popularly known as NS-2 [1], is a discrete event network simulator in which each event occurs at an instant in time. NS is an object oriented simulator, written in C++, with an OTcl interpreter as a frontend. It is used to simulate routing and multicast protocols. It supports large number of network protocols for simulation and provides results for wired, wireless and wired-cum-wireless scenarios. NAM (Network AniMator) is the most important feature of NS. It gives the visual outputs of the simulation scenarios. As NS is open source and plentiful documentation is available, it is very popular for simulations and extensions.

3.2 NS-2 Architecture

NS uses two different languages because the simulator has two different kinds of things it needs to do. First, detailed simulation of protocol requires system programming language which can efficiently manipulate bytes, packet headers and implement algorithms that run over large data sets. For these tasks, run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important. On the other hand, a large part of network research involves slightly varying parameters or configurations or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important.

NS meets both of these needs with two languages, C++ and OTcl [17]. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration [17]. NS (via tclcl) provides glue to make objects and variables appear on both languages [6].

3.3 Installation of NS2

3.3.1 Installation of Fedora

The various platforms supported for NS are Linux, Solaris and Windows XP using Cygwin. We have used Fedora core 8 (Linux environment). Following are the steps to install Fedora core 8 OS [8]. Fedora core 8 is freely available at [7].

- Remove data from one of the drives where fedora is to be installed.
- Load the CD in drive.
- Select GUI mode by pressing enter.
- Skip for disc checking.
- Then select the English US language.
- Select Custom layout for partitioning hard disk and click next.
- Where we want to install, make that disk space free first (at least 5GB). The drives are named as dev/sda1 (which is C drive in the system) and so on for the other drives.
- Then select New for that free space.
- Give a mount point as '/'(root) & select ext3 & give a space equal to total free space minus 2*RAM. This space is given in MB.
- Then make it ok.
- Then on the remaining free space select New & then select file format as Swap Give space to swap 2*RAM size. This space is to be given in MB.
- Select grub boot loader & make windows or other OS as by default.
- Select all the software if needed.
- Give the computer name
- Select region as Asia/Calcutta.
- Enter Root password
- Then installation will start (1103 packet should be installed for proper functioning of fedora).

3.3.2 Installation of NS-2 (ns-2.30)

The installation procedure of NS-2 in Fedora core 8 OS is as follows. It should be installed in the root account of the fedora. NS-2 is freely available at [9].

- Go to link: SourceForge.net: Files.
- Download file: ns-allinone-2.30.tar.gz
- Open the root account in Fedora.
- Extract the downloaded file on desktop.
- Open console and give following commands.
  - cd Desktop
  - cd ns-allinone-2.30
  - ./install
- Now, the installation will start and after some time the message named “IMPORTANT NOTICES” will appear on screen which gives the information about the paths to be added.

  (1) You MUST put /home /myusername /ns-allinone-2.29 /otcl-1.11, /home / myusername /ns-allinone-2.29 /lib, into your LD_LIBRARY_PATH environment variable.

  If it complains about X libraries, add path to your X libraries into LD_LIBRARY_PATH.

  If you are using csh, you can set it like: setenv LD_LIBRARY_PATH <paths>

  If you are using sh, you can set it like: export LD_LIBRARY_PATH=<paths>

  (2) You MUST put /home/myusername/ns-allinone-2.29/tcl8.4.11/library into your TCL_LIBRARY environmental variable. Otherwise ns/nam will complain during startup.

(3) [OPTIONAL] To save disk space, you can now delete directories tcl8.4.11 and tk8.4.11. They are now installed under /home/myusername/ns-allinone-2.29 / {bin,include,lib}

After these steps, you can now run the ns validation suite with cd ns-2.29; ./validate

For trouble shooting, please first read ns problems page http://www.isi.edu/nsnam/ns/ns-problems.html. Also search the ns mailing list archive for related posts.

- Give the following command in already opened console.
  - gedit ~/.bashrc
- Then the "gedit" text editor will open which is the official text editor for the GNOME (GNU Object Model Environment) desktop.
- Keep the written things in that editor as it is and add the following lines at the end of it. Replace /usr/local/" by installation path like "/root/Desktop". Also, make changes according to the version of the ns. For example, for ns-2.30, otcl's version is 1.12, tcl's version is 8.4.13, etc.

```
#LD_LIBRARY_PATH
OTCL_LIB=/usr/local/ns-allinone-2.30/otcl-1.12
NS2_LIB=/usr/local/ns-allinone-2.30/lib
X11_LIB=/usr/X11R6/lib
USR_LOCAL_LIB=/usr/local/lib
export LD_LIBRARY_PATH = $LD_LIBRARY_PATH: $OTCL_LIB: $NS2_LIB: $X11_LIB: $USR_LOCAL_LIB
# TCL_LIBRARY
TCL_LIB=/usr/local/ns-allinone-2.30/tcl8.4.13/library
USR_LIB=/usr/lib
export TCL_LIBRARY=$TCL_LIB:$USR_LIB

# PATH
XGRAPH=/usr/local/ns-allinone-2.30/bin:/usr/local/ns-allinone-2.30/tcl8.4.13/unix:/usr/local/ns-
```

allinone-2.30/tk8.4.13/unix

NS=/usr/local/ns-allinone-2.30/ns-2.30/

NAM=/usr/local/ns-allinone-2.30/nam-1.12/

PATH=$PATH:$XGRAPH:$NS:$NAM

- Now, give the following command in the already opened console.
  - source ~/.bashrc
- Reopen the console now.
- Write ns and press enter. If there is no error, a "%" will appear on screen.
- Press exit to quit this mode or back to prompt

3.4 Simulation

Following are the steps to write tcl file to in NS which is necessary to run the simulation.

- Define goal and expected result
- Create network topology
  - Nodes
  - Link (simplex/duplex, BW, delay, etc.) [18]
- Specify agents (TCP, UDP, etc.) [18].
- Traffic sources (CBR) [18].
- Simulation scenario (wired, wireless, wired-cum-wireless)

3.4.1 TCL file

As the project is on wireless protocol, the simple example of simulation in wireless scenario is explained below.

set val(chan) Channel/WirelessChannel ;# Channel type

```
set val(prop)             Propagation/TwoRayGround          ;# Radio-propagation model
set val(ant)              Antenna/OmniAntenna               ;# Antenna type
set val(ll)               LL                                ;# Link layer type
set val(ifq)              Queue/DropTail/PriQueue           ;# Interface queue type
set val(ifqlen)           50                                ;# Max packet in ifq
set val(netif)            Phy/WirelessPhy                   ;# Network interface type
set val(mac)              Mac/802_11                        ;# MAC type
set val(nn)               4                                       ;# Number of mobile nodes
set val(rp)               AODV                              ;# Routing protocol
set val(energymodel)  EnergyModel                           ;# Energy model
set val(initialenergy)    initialEnergy                     ;# Initial energy in Joules
set val(x)                500                               ;#Set simulation area
set val(y)                500


set ns [new Simulator]                                      ;#Create a simulator object
$ns use-newtrace                                            ;#For new wireless trace format
set f [open out.tr w]                                       ;#Open general trace file
$ns trace-all $f
set namtrace [open out.nam w]                               ;#Open nam trace file
$ns namtrace-all-wireless $namtrace $val(x) $val(y)
set topo [new Topography]                                   ;#Create a topography and
$topo load_flatgrid 800 800                                 ;#Define it in 800x800 area
create-god $val(nn)                                         ;#Create “God”
$ns node-config  -adhocRouting $val(rp) \                   ;#Configure and create nodes
-llType $val(ll) \
-macType $val(mac) \
-ifqType $val(ifq) \
-ifqLen $val(ifqlen) \
-antType $val(ant) \
-propType $val(prop) \
-phyType $val(netif) \
-topoInstance $topo \
 -agentTrace OFF \                                          ;#Keep trace ON or OFF     as
                                                           needed
-routerTrace ON \
-macTrace ON \
-movementTrace OFF \
 -energyModel $val(energymodel) \                           ;#Creating energy model
 -txPower 0.3\                                              ;#Define transmission power
 -rxPower 0.2\                                              ;#Define receiver power
-initialEnergy $val(initialenergy)\
 -initialEnergy 100                                         ;#Define initial energy
```

```
proc finish {} {                                        ;#Finish procedure
                global ns f namtrace
                $ns flush-trace
                close $namtrace
                exec nam -r 5m out.nam &
                exit 0
 }
 set node [$ns_ node]                                   ;#Create a mobile node
 $node random-motion 0                                  ;#Disable random motion
 for  {set i < 0}  {$i<4}  {incr i}                     ;#Use for to create 4 nodes
          {
          set node_($i) [$ns_ node]
          }
 for {set i 0} {$i < $val(nn)} {incr i}                 ;#Initial node position
          {
          $ns initial_node_pos $n($i) 30+i*100
          }
 $ns at 0.0 "$n(0) setdest 150.0 100.0 3000.0"          ;#Create node movement
 $ns at 0.0 "$n(1) setdest 250.0 200.0 3000.0"
 $ns at 0.0 "$n(2) setdest 400.0 200.0 3000.0"
 $ns at 0.0 "$n(3) setdest 550.0 200.0 3000.0"

 set sink0 [new Agent/LossMonitor]                      ;# Configure and set up a flow
 set sink1 [new Agent/LossMonitor]
 set sink2 [new Agent/LossMonitor]
 set sink3 [new Agent/LossMonitor]
 $ns attach-agent $n(0) $sink0                          ;#Attach nodes to sink
 $ns attach-agent $n(1) $sink1
 $ns attach-agent $n(2) $sink2
 $ns attach-agent $n(3) $sink3
 set tcp0 [new Agent/TCP]                               ;#Define tcp for traffic generation
 set tcp1 [new Agent/TCP]
 set tcp2 [new Agent/TCP]
 set tcp3 [new Agent/TCP]
 $ns attach-agent $n(0) $tcp0                           ;# Attach nodes to tcp source
 $ns attach-agent $n(1) $tcp1
 $ns attach-agent $n(2) $tcp2
 $ns attach-agent $n(3) $tcp3
 proc attach-CBR-traffic { node sink size interval } {
 set ns [Simulator instance]                            ;#Get an instance of the simulator
 set cbr [new Agent/CBR]                                ;#Create a CBR  agent and attach it
 $ns attach-agent $node $cbr
 $cbr set packetSize_ $size
```

```
$cbr set interval_ $interval
$ns connect $cbr $sink                                  ;#Attach CBR source to sink;
return $cbr
  }
set cbr0 [attach-CBR-traffic $n(0) $sink1 1000 .015]
set cbr1 [attach-CBR-traffic $n(1) $sink2 1000 .015]
set cbr2 [attach-CBR-traffic $n(2) $sink3 1000 .015]
set cbr3 [attach-CBR-traffic $n(3) $sink0 1000 .015]
$ns at 0.0 "record"
$ns at 0.5 "$cbr1 start"
$ns at 1.0 "$cbr0 start"
$ns at 1.5 "$cbr3 start"
$ns at 4.0 "$cbr3 stop"
$ns at 5.0 "$cbr0 stop"

$ns at 10.0 "finish"
puts "Start of simulation.."
$ns run
```

3.4.2 Steps to run simulation

To run the tcl file, written in text editor, type the following command in the console.

- ns filename.tcl

When we give this command, ns does the following procedure.

- Reads the tcl file
- Runs the simulation program
- Creates trace files (general trace file as well as NAM file)

3.4.3 Trace file format

In wireless simulations, there are two trace formats, old and new. New trace format can be produced by calling the command "$ns use-newtrace" before the universal trace command. As new trace file format [10] is very descriptive and useful for analyzing various important parameters, it is discusses below with simple example.

s -t 0.267662078 -Hs 0 -Hd -1 -Ni 0 -Nx 5.00 -Ny 2.00 -Nz 0.00 -Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 0.255 -Id -1.255 -It message -Il 32 -If 0 -Ii 0 -Iv 32

s -t 1.511681090 -Hs 1 -Hd -1 -Ni 1 -Nx 390.00 -Ny 385.00 -Nz 0.00 -Ne -1.000000 -Nl RTR -Nw --- -Ma 0 -Md 0 -Ms 0 -Mt 0 -Is 1.255 -Id -1.255 -It message -Il 32 -If 0 -Ii 1 -Iv 32

Explanation of the format of above example is given below.

Event type

s – Send

r – Receive

f – Forward

General tag

-t 0.267662078 - Time

Node property tags

-Ni 0 - Node id

-Nx 5.00 - Node's x-coordinate

-Ny 2.00 - Node's y-coordinate

-Nz 0.00 - Node's z-coordinate

-Ne 1.00 - Node energy level

-Nl RTR - Trace level, such as AGT, RTR, MAC

-Nw - Reason for the event. The different reasons for dropping a packet are given below.

"END" DROP_END_OF_SIMULATION

"COL" DROP_MAC_COLLISION

"DUP" DROP_MAC_DUPLICATE

"ERR" DROP_MAC_PACKET_ERROR

"RET" DROP_MAC_RETRY_COUNT_EXCEEDED

"STA" DROP_MAC_INVALID_STATE

"BSY" DROP_MAC_BUSY

"NRTE" DROP_RTR_NO_ROUTE i.e no route is available.

"LOOP" DROP_RTR_ROUTE_LOOP i.e there is a routing loop

"TTL" DROP_RTR_TTL i.e TTL has reached zero.

"TOUT" DROP_RTR_QTIMEOUT i.e packet has expired.

"CBK" DROP_RTR_MAC_CALLBACK

"IFQ" DROP_IFQ_QFULL i.e no buffer space in IFQ.

"ARP" DROP_IFQ_ARP_FULL i.e dropped by ARP

"OUT" DROP_OUTSIDE_SUBNET i.e dropped by base stations on receiving routing updates

Packet information at IP level

-Is 0.255 - Source address.source port number

-Id 1.255 - Dest address.dest port number

-It message - Packet type

-Il 32 - Packet size

-If 0 - Flow id

-Ii 0 - Unique id

-Iv 32 - Ttl value

Next hop info

-Hs 0 - Id for this node

-Hd 1- Id for next hop towards the destination.

Packet info at MAC level

-Ma 0 - Duration

-Md 0 - Dst's ethernet address

-Ms 0 - Src's ethernet address

-Mt 0 - Ethernet type

Packet info at "Application level"

-P arp Address Resolution Protocol. Details for ARP is given by the following tags:

-Po: ARP Request/Reply

-Pm: Src mac address

-Ps: Src address

-Pa: Dst mac address

-Pd: Dst address

-P dsr This denotes the adhoc routing protocol called Dynamic source routing. Information on DSR is represented by the following tags:

-Pn: How many nodes traversed

-Pq: Routing request flag

-Pi: Route request sequence number

-Pp: Routing reply flag

-Pl: Reply length

-Pe: Src of srcrouting->dst of the source routing

-Pw: Error report flag ?

-Pm: Number of errors

-Pc: Report to whom

-Pb: Link error from linka->linkb

-P cbr Constant bit rate. Information about the CBR application.

-Pi: Sequence number

-Pf: How many times this pkt was forwarded

-Po: Optimal number of forwards

-P tcp Information about TCP flow is given by the following subtags:

-Ps: Seq number

-Pa: Ack number

-Pf: How many times this pkt was forwarded

-Po: Optimal number of forwards

3.4.4 NAM (Network AniMator)

Network Animator is a visualization (animation) tool of ns. It shows network simulation traces and real world packet traces. It supports topology layout, nodes, links, queues, node connectivity, packet level animation and various data inspection tools. When the simulation completes, it will attempt to run NAM visualization of the simulation on the screen if we have created namtrace file [11].

Figure 3.1 shows the window of NAM and description of various tools available in it [11]

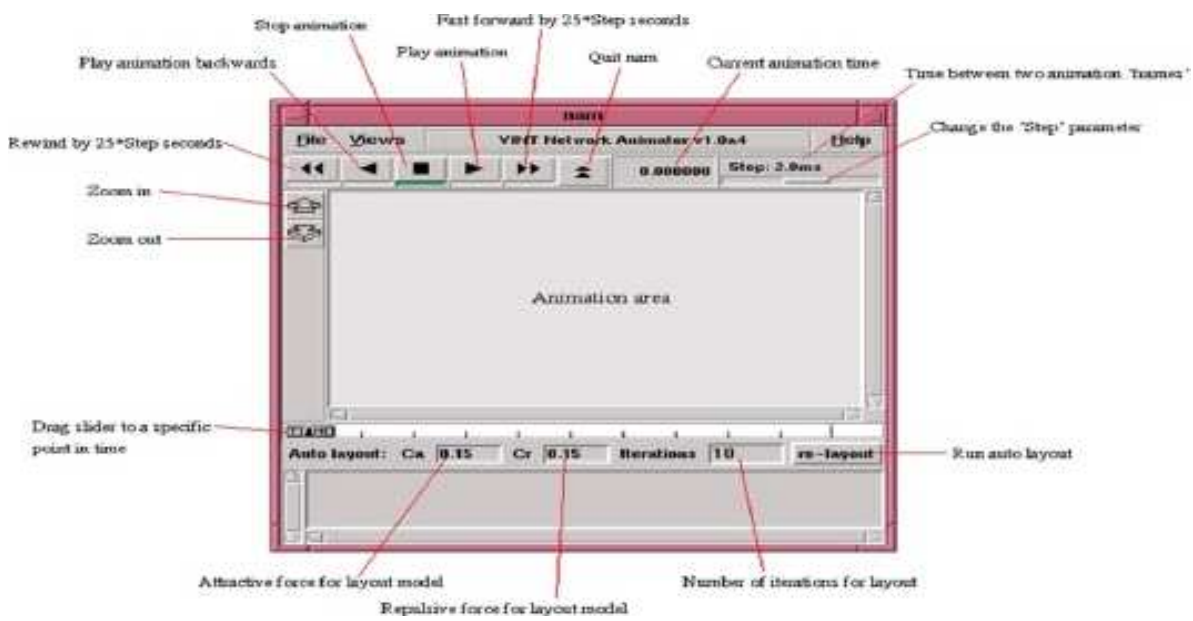

Fig 3.1 NAM window

3.4.5 Trace analysis using AWK

Awk actually stands for the names of its authors: "Aho, Weinberger, & Kernighan". An Awk is a text processing language which is used for creating small databases, creating reports from the results, performing mathematical operations on files of numeric data. Awk, indeed, is a filter programming language for performing complex text-processing tasks. Awk programs can be entered on the command line or we can implement complicated programs containing dozens of lines of Awk statements. Anybody who can write C program can use Awk with little difficulty. Although, C and Awk have significant differences beyond their many similarities [12].

Suppose, we have the file named “throughput.tr” that describes number of packets sent and received for different number of nodes as shown below.

0	12	11
1	17	15

2	1	1
3	8	8
4	23	20
5	15	12

Here, first column describes the node id, second describes number of packets sent by that node and third column shows number of packets received at the destination i.e. base station. Now, it is required to find throughput from this data. For that, awk file should be made as follows. It is named as throughput.awk

```
BEGIN {
        recv = 0;
        send = 0;
}{
        send = send + $2;
        recv = recv + $3;
}
END {
        printf(" Packets sent =  %d", send);
        printf(" \n Packets received =  %d", recv);
        printf(" \n Throughput =  %f  %", recv/send*100);
}
```

To run this awk file, following awk command has to be executed in console.

- Awk -f throughput.awk throughput.tr > throughput.log

This command will produce the output in the new file named throughput.log as follows.

Packet sent = 76

Packets received = 67

Throughput = 88.15 %

3.4.6 Plotting using GNU plot

GNU plot is a command driven visualization tool for interactive function plotting. It is used to plot the results in a way we want. It supports multiple platforms like Linux/Unix, Macs and lot of others. It has a very small set of commands which are very easy to learn.

Installation of Gnuplot

Following are the steps to install Gnuplot in fedora core 8 [13]. It is available at [13].

- Download the Gnuplot and extract it on the desktop.
- Run the following commands in console.
  - cd Desktop
  - cd gnuplot-4.2.4
  - ./configure
  - make
  - make install

To plot the results using Gnuplot, we must have two files on hand. Text file and log file. Text file contains the executable statements to provide various parameters needed and log file contains the data to be plotted. Following is the example of text file, named compare.txt.

Create a title: > set title "Comparing the performance of ADTCP and Improved- ADTCP"

For color or mono: > set term post enh color/mono

Create file: > set out 'comparision.eps'

Put a label on the y-axis: > set xlabel "Total Transmitted Packets (pkts)"

Put a label on the x-axis: > set ylabel "Number of Concurrent TCP Flows"

Change the x-axis range: > set xrange [0:30]

Change the y-axis range: > set yrange [210000:310000]

Change the tic-marks: > set xtics 5

Change the tic-marks: > set ytics 10000

Data to be plotted: > plot "compare.log" using 1:2 title "Improved-ADTCP" with linespoints, "compare.log" using 1:3 title "ADTCP" with linespoints

Following is the example of the simple log file, named compare.log, related to the text file described above.

5	263213	239261
10	285249	262446
15	285341	262813
20	299212	278267
25	302492	284342

To plot the results from the log file, following commands have to be executed in console.

- gnuplot
- load “compare.txt”

The Gnuplot of the above example will look like as shown below.

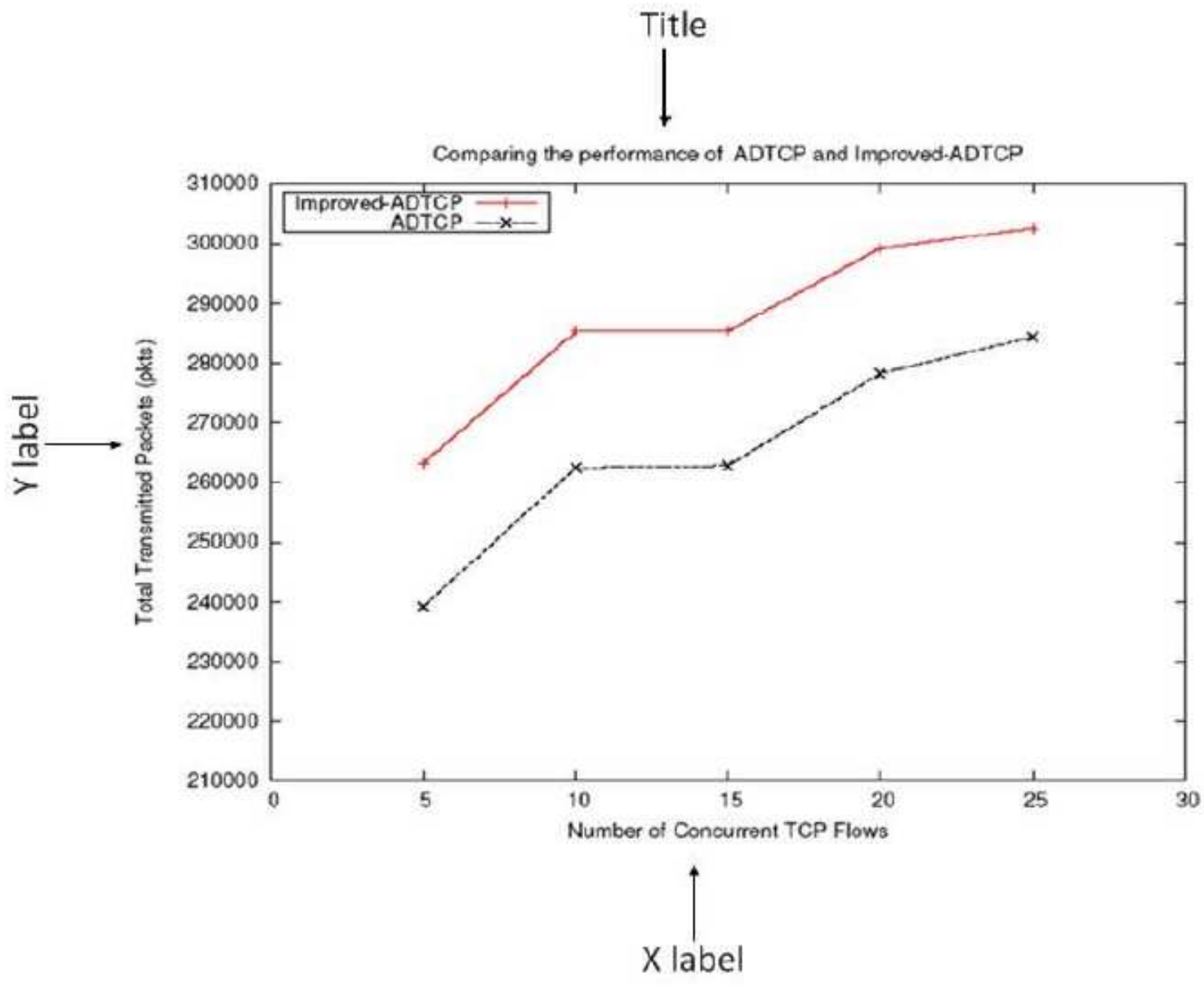

Fig 3.2 GNU graph

Chapter 4

Simulating S-MAC using Network Simulator-2 (NS-2)

4.1 S-MAC in NS-2

4.1.1 Implemented S-MAC Features

We have studied the IEEE paper[1], In this paper S-MAC is developed on the Mote platform that runs TinyOS and its implementation in TinyOS reflects the latest protocol development. But as a hardware implementation, which needs real hardware as its running platform. S-MAC has been also implemented in ns-2, the network simulator. The latest official release is integrated in ns-2.34. If you want to use the most recent updated and debugged version, you have to download a new ns-2. More information about the updating of ns-2 and S-MAC can be found on the ns-2 homepage [13].

In this report, we study S-MAC with ns-2 and install the ns-2.34 all-in-one version on the Fedora core 8, Linux system. After installation, the source files (in C++) for S-MAC, smac.h and smac.cc, can be found in the path ns-2.28\mac\.

We list all features of S-MAC that have been implemented in ns-2-34 as follows. The detailed descriptions of these features have been presented in chapter 2.

Basic features

Listen and sleep periodically according to schedules

Both physical and virtual carrier sense

Overhearing avoidance

RTS/CTS mechanism for unicast data packets

Synchronization algorithm (including schedule choosing, schedule updating and maintaining)

4.1.2 S-MAC Parameters Settings

All preset parameters for S-MAC can be found and modified in the header file smac.h, including user adjustable S-MAC parameters, internal S-MAC parameters, and physical layer parameters.

Some other parameters will be assigned at the start of the simulation run, because their values depend on some interactive parameters. For example, the sleep time and cycle time (frame time) depend on the value of duty cycle that users have set in the tcl script. We call those parameters, which can be set in tcl scripts in an interactive way, interactive parameters. All three interactive parameters are listed in Table 4.1 below.

Table 4.1. S-MAC Interactive Parameters

Name	Comment
syncFlag_	If it is set to 1, S-MAC runs with periodic sleep. If it is set to 0, S-MAC runs without periodic sleep.
dutyCycle_	The value of duty cycle in percent. It controls the length of sleep. If not set, ns-2 uses the default value 10%. This parameter is active only when syncFlag_ is set to 1.
selfConfigFlag_	If it is set to 1, all S-MAC nodes follow the schedule initialization algorithm described in section 2.3. If it is set to 0, the schedule start time (first listen period start time) for each node is user-configurable.

4.1.3 S-MAC Modes

For the purpose of comparison, S-MAC can be configured to run under three modes, which are listed as below:

Mode 1: S-MAC without periodic sleep

If the parameter syncFlag_ is set to 0, S-MAC will work under this mode. In this mode, each node does not follow periodic sleep and listen cycle and runs in a fully active mode.

Mode 2: S-MAC with periodic sleep

If the parameters syncFlag_ is set to 1. S-MAC will work in this mode. In this mode, only the basic features listed in sub-section 4.1.1 are available. No adaptive listening and message passing are available in this mode.

Mode 3: S-MAC different periodic sleep cycle.

If the duty cycle is vary then its listen time also vary in this project we made a one graph which will compare the 2 different dutycycle.

4.2 Preparations for Simulating S-MAC

In this section, we introduce what modules we have added to ns-2 necessary for simulating S-MAC. An Example Tcl Script for Simulating S-MAC .We have getting a TCL script which will be made by Vijay Kakadia during his research project under the guidance of Dr. Wei Ye. Padma Haldar provided insights about *ns*-2 internals and integrated Vijay's code into *ns*-2. The code is checked into *ns*-2 on June 14, 2005. We have take a base of this code understand the code and compare the two protocol using this code.

Topology: we have create 2 node topology in which one node is sender and other is receiver. For our 3 graphs we take this part. For routing node graph we have created a 3 node topology in which one node is simply routing node and get our last graph at a routing node.

Traffic pattern: Packets are sent from the source node to the sink node. Here we used CBR traffic and change the “packet inter arrival time” from 0.1s to the 10s. Here 0.1s means high traffic and 10s means low traffic and see how s-mac can behave on 2 different conditions.

4.2.1 Trace Format

All trace data will be recorded into a specified trace file in a certain format. Ns-2 defines two trace formats, old format and new format. The old format is the default trace format in ns-2 and supports all type of simulations. The new format is especially designed for tracing wireless simulations, which is more standardized than the old one. However, S-MAC in ns-2-28 still does not support the new trace format. Therefore, we consider the old format. A trace record in a S-MAC trace file is shown in Fig. 4..3 below.

s 274.090231333 _3_ MAC --- 0 RTS 10 [0.11 4 3] 123456789

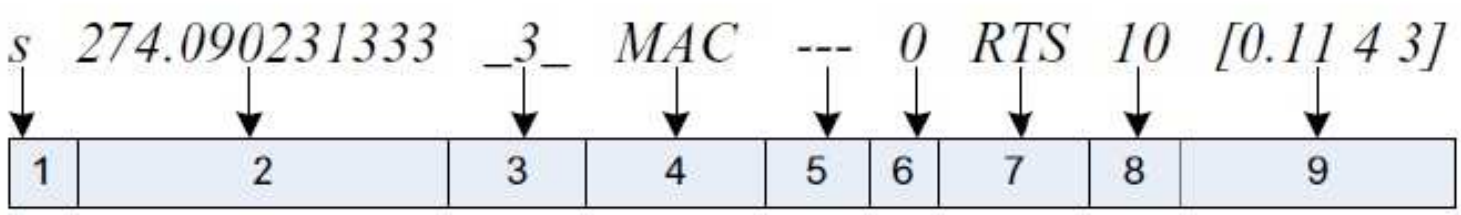

Fig. 4.1. S-MAC Trace format

We can know from this trace record that at 274.090231333 s, node 3 sends an RTS packet to node 4. Now we explain what these field mean in the usual case.

1. The first field represents the event type, whose value can be s (send), r (receive), d (drop) or f (forward).
2. The second field the time this event happens.
3. The third field records the id of the node, on which this event takes place.
4. The fourth field shows the layer where this event happens. Its possible value may be one of the following four: AGT (agent), RTR (router), IFQ (interface queue), and MAC (mac).
5. The fifth field is normally a short broken line, which is reserved for special events. For example, when collision occurs, the broken line is replaced with COL.
6. The sixth field is the global sequence number for this packet, which is the integer number used to identify this packet in the whole network and distinguish it from others. Sequence number is only available for data packets and not allocated for control packets, like RTS/CTS/ACK and SYNC packets in S-MAC (using a zero instead).
7. The seventh field is the packet type. The actual value is determined by the application or MAC layer, which creates this packet. For example, cbr represents that it is a data packet generated by a CBR traffic source.
8. The eighth field is the packet size in bytes.
9. The ninth field including three numbers in brackets concerns MAC layer information. Originally, there will be four numbers in the brackets. But S-MAC revises this format. The first number is the duration field of this packet. In Fig. 4.3, the duration field of this

RTS packet is 0.11 s, which is the remaining time reserved for the coming transmission. The second number stands for the MAC address of the receiver of this packet, and the third number for the sender.

The above nine fields are common for all traces if S-MAC is employed. But some packets may have additional fields to record other information, like routing and IP information. But for our purpose of study, considering these nine fields are already enough.

Chapter 5

Simulation Results

In this chapter, we present our simulation results after studying S-MAC.

- **Measurement of energy consumption:**

- First we run the s-mac program
- Then run the 802.11 program
- Form the trace file we compare the ideal energy of both using the awk script.
- Because for s-mac periodic sleep and listen is employed so, obviously energy consumption of s-mac is low compared to the 802.11
- The Gnuplot of these are below:

Fig. 5.1. Measurement of Energy Consumption

- **Mesurement of delay vs.energy consumption:**

- Here we change the duty cycle of the s-mac
- We take first duty cycle 10%
- For the second case we have taken it 20%
- So, basically we changed the listen period of the s-mac
- We get the result that as a listen time is small energy consumption is less and delay is more.

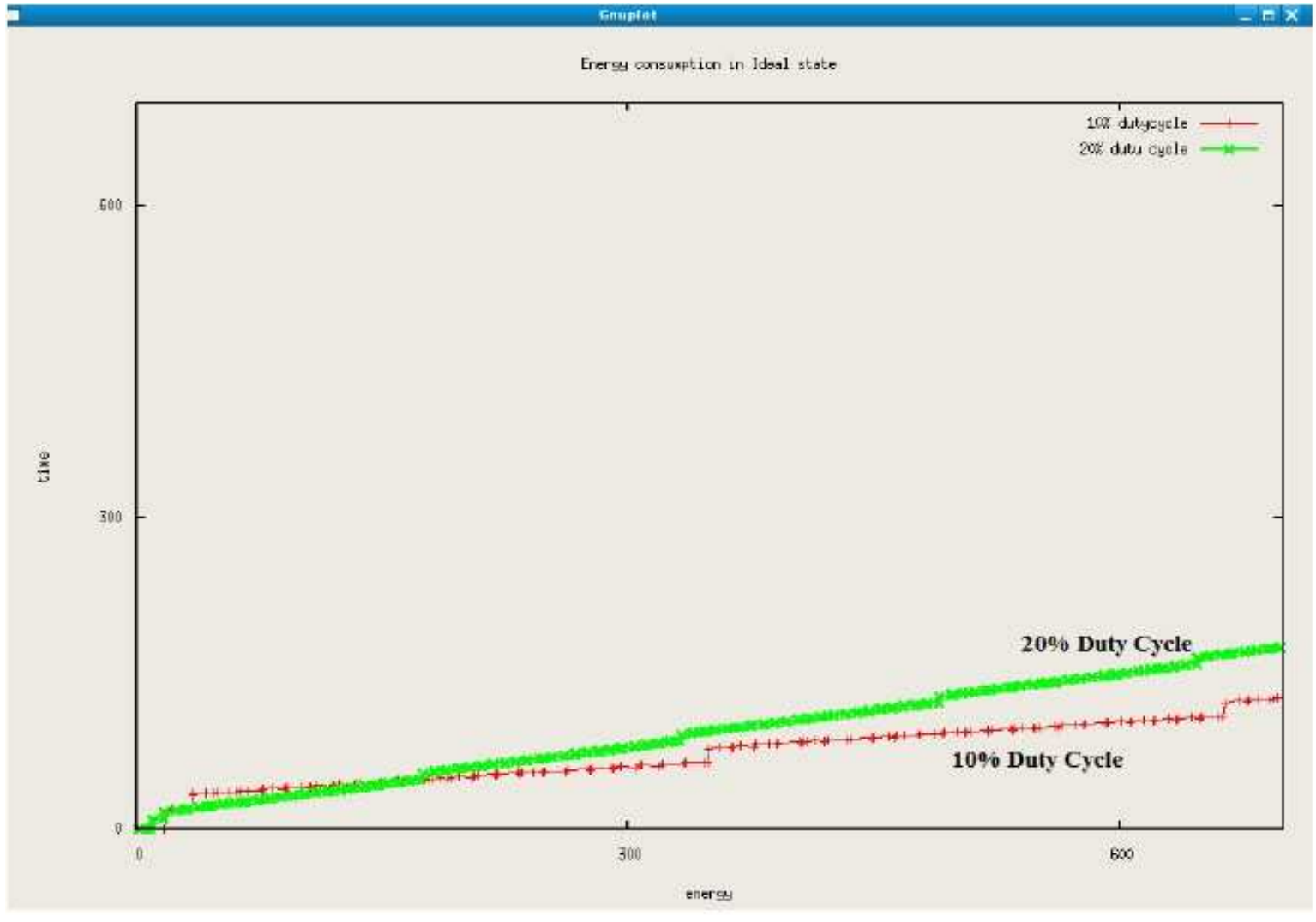

Fig. 5.2. Measurement of Delay Vs Energy

- **Traffic variations vs. energy consumption :**

- Here for the traffic variation we change the CBR traffic interval from 0.001 to 10s.
- Here 0.001s is high traffic case and 10s is the low traffic case.
- We see that for s-mac there is no such variation for high or low traffic means s-mac will adjust its sleep period according to the situation.
- While for 802.11 energy consumption is large.

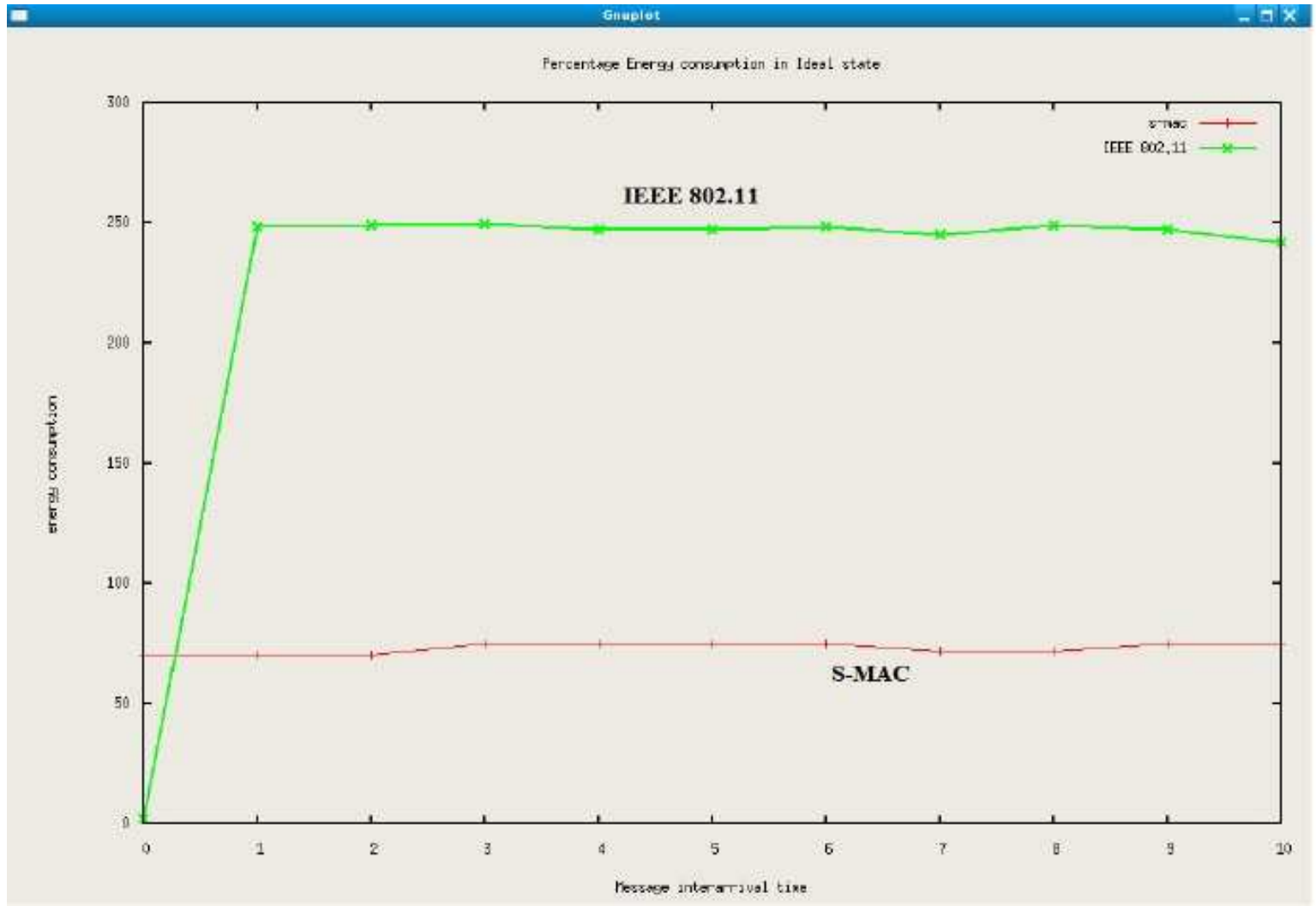

Fig. 5.3 Measurement of Traffic Variation Vs Energy Consumption

➢ **Energy consumption at intermediate node :**

- Here we design a 3 node topology
- 1 node are sender and 1 node are sink and 1 is router.
- Here we compare the ideal energy state of the intermediate node.
- For s-mac variation of energy consumption at an intermediate node is very less while for 802.11 it is large variation.
- See the graph

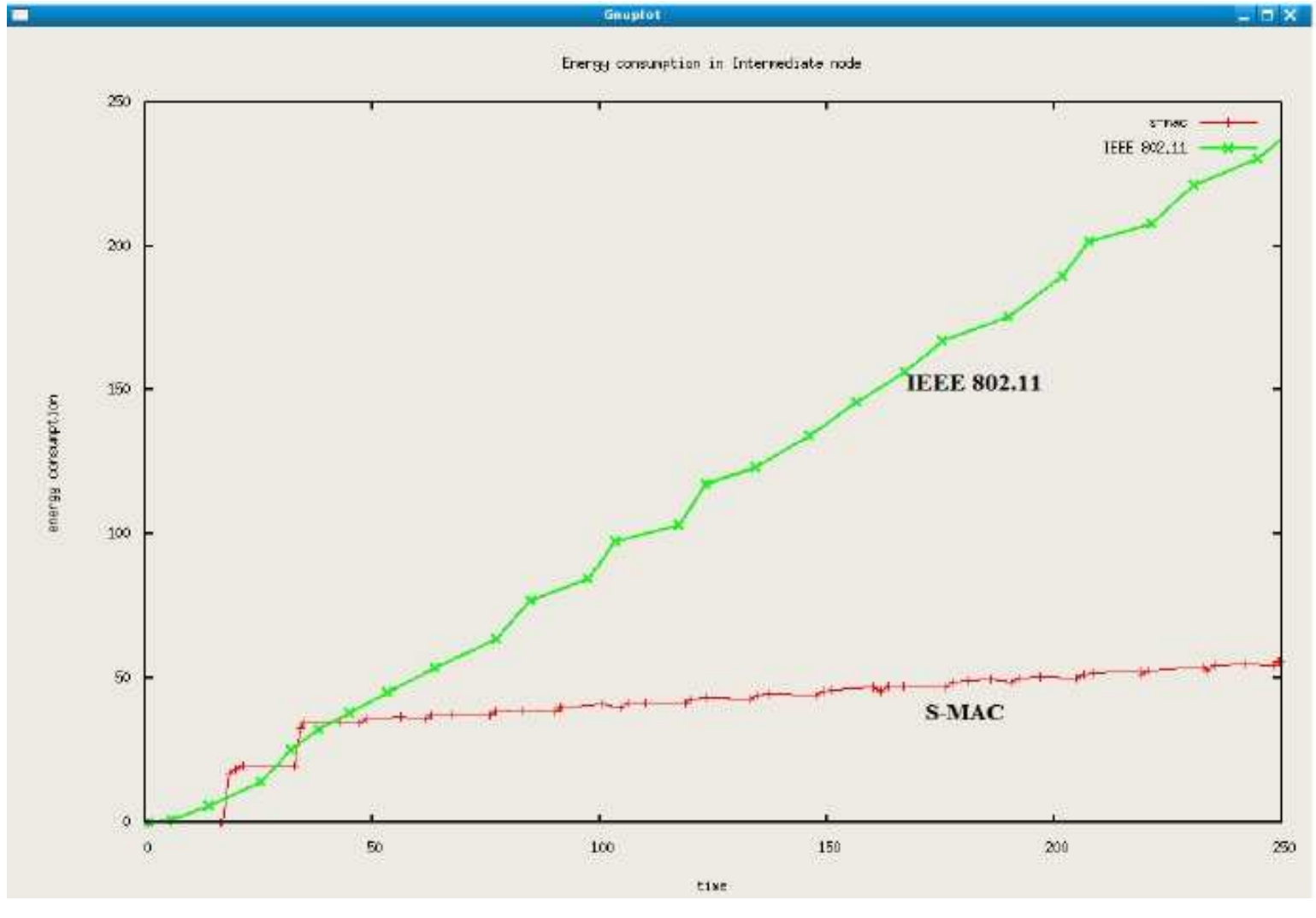

Fig. 5.4 Measurement of Energy Consumption at Intermediate Node

Chapter 6

Conclusion

During the project work, we have done the following two main tasks.

1. We have simulate S-MAC in ns-2.34 and found out how S-MAC works internally. Some important features of S-MAC, such as message passing, overhearing avoidance etc.

2. We have compare the s-mac protocol with the IEEE 802.11.

Through analyzing the simulation results, we have the following conclusions:

S-MAC with periodic sleep efficiently reduces the energy consumptions due to idle listening. However, periodic sleeping increases latency and reduces throughput. So,s-mac is a trade off between energy and delay.

Appendix A

TCL File :

```
set opt(chan)           Channel/WirelessChannel
set opt(prop)           Propagation/TwoRayGround
set opt(netif)          Phy/WirelessPhy
set opt(mac)            Mac/802_11                  ;# MAC type
set opt(mac)            Mac/SMAC                    ;# MAC type
set opt(ifq)            Queue/DropTail/PriQueue
set opt(ll)             LL
set opt(ant)            Antenna/OmniAntenna

set opt(x)              800                         ;# X dimension of the topography
set opt(y)              800                         ;# Y dimension of the topography
set opt(cp)             "../mobility/scene/cbr-50-10-4-512"
set opt(sc)             "../mobility/scene/scen-670x670-50-600-20-0"

set opt(ifqlen)         50                          ;# max packet in ifq
set opt(nn)             2                           ;# number of nodes
set opt(seed)           0.0
set opt(stop)           700.0                       ;# simulation time
set opt(tr)             MyTest.tr                   ;# trace file
set opt(nam)            MyTest.nam                  ;# animation file
set opt(rp)             DumbAgent                   ;# routing protocol script
set opt(lm)              "off"                      ;# log movement
set opt(agent)          Agent/DSDV
```

```
set opt(energymodel)    EnergyModel
#set opt(energymodel)    RadioModel
set opt(radiomodel)      RadioModel
set opt(initialenergy)    1000                ;# Initial energy in Joules
#set opt(logenergy)       "on"                ;# log energy every 150 seconds


Mac/SMAC set syncFlag_ 1

Mac/SMAC set dutyCycle_ 10

set ns_          [new Simulator]
set topo         [new Topography]
set tracefd      [open $opt(tr) w]

set prop         [new $opt(prop)]

$topo load_flatgrid $opt(x) $opt(y)
ns-random 1.0
$ns_ trace-all $tracefd
#$ns_ namtrace-all-wireless $namtrace 500 500


# Create god

create-god $opt(nn)


#global node setting

      $ns_ node-config -adhocRouting DumbAgent \
                       -llType $opt(ll) \
                       -macType $opt(mac) \
                       -ifqType $opt(ifq) \
                       -ifqLen $opt(ifqlen) \
                       -antType $opt(ant) \
                       -propType $opt(prop) \
                       -phyType $opt(netif) \
                       -channelType $opt(chan) \
                       -topoInstance $topo \
                       -agentTrace ON \
                       -routerTrace ON \
                       -macTrace ON \
```

```
                        -energyModel $opt(energymodel) \
                        -idlePower 1.0 \
                        -rxPower 1.0 \
                        -txPower 1.0 \
                       -sleepPower 0.001 \
                    -transitionPower 0.2 \
                    -transitionTime 0.005 \
                    -initialEnergy $opt(initialenergy)

$ns_ set WirelessNewTrace_ ON
#set AgentTrace                 ON
#set RouterTrace                OFF
#set MacTrace                   ON

        for {set i 0} {$i < $opt(nn) } {incr i} {
                set node_($i) [$ns_ node]
                $node_($i) random-motion 0          ;# disable random motion
        }

#       $node_(1) set agentTrace ON
#       $node_(1) set macTrace ON
#       $node_(1) set routerTrace ON
#       $node_(0) set macTrace ON
#       $node_(0) set agentTrace ON
#       $node_(0) set routerTrace ON

set udp_(0) [new Agent/UDP]
$ns_ attach-agent $node_(0) $udp_(0)
set null_(0) [new Agent/Null]
$ns_ attach-agent $node_(1) $null_(0)
set cbr_(0) [new Application/Traffic/CBR]
$cbr_(0) set packetSize_ 512
$cbr_(0) set interval_ 10.0
$cbr_(0) set random_ 1
$cbr_(0) set maxpkts_ 50000
$cbr_(0) attach-agent $udp_(0)
$ns_ connect $udp_(0) $null_(0)



$ns_ at 1.00 "$cbr_(0) start"
#$ns_ at 177.000                "$node_(0) set ifqLen"
```

```
# Tell all the nodes when the simulation ends

for {set i 0} {$i < $opt(nn) } {incr i} {
    $ns_ at $opt(stop) "$node_($i) reset";
}
$ns_ at $opt(stop) "puts \"NS EXITING...\" ; $ns_ halt"

set b [$node_(0) set mac_(0)]
#set c [$b set freq_]
set d [Mac/SMAC set syncFlag_]

#set e [$node_(0) set netif_(0)]

#set c [$e set L_]
set c [Mac/SMAC set dutyCycle_]
#puts $tracefd "M 0.0 nn $opt(nn) x $opt(x) y $opt(y) rp $opt(rp)"
#puts $tracefd "M 0.0 sc $opt(sc) cp $opt(cp) seed $opt(seed)"
#puts $tracefd "M 0.0 prop $opt(prop) ant $opt(ant)"
#puts $tracefd "V $b : $c : $d :"
puts "Starting Simulation..."
$ns_ run
```

References

[1] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks.

[2] W. Ye, J. Heidemann and D.Estrin. Medium access control with coordinated, adpative sleeping for wireless sensor networks. *ACM/IEEE Transactions Networking*, 12(3):493-506, June 2004.

[3] Y. Li, W. Ye and J. Heidemann. Energy and latency control in low duty cycle MAC protocols. *USC/ISI Technical Report ISI-TR-595*, August 2004.

[4] A. M. Law, W. D. Kelton. Simulation, Modeling and Analysis, 3rd Edition.
McGraw-Hill, [2000].

[5] L. Bao and J. J. Garcia-Luna-Aceves. A New Approach to Channel Access Scheduling for Ad Hoc Networks. In *Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking*, pages 210–220,Rome, Italy, July 2001. ACM.

[6] V. Kanodia, C. Li, A Sabharwal, B. Sadeghi, and E. W. Knightly. Distributed Multi-Hop Scheduling and Medium Access with Delay and Throughput Constraints. In *Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking*, pages 200–209, Rome, Italy, July 2001. ACM.

[7] C. Schurgers, V. Tsiatsis, S. Ganeriwal, and M. B. Srivastava. Optimizing Sensor Networks in the Energy-Latency-Density Design Space. *IEEE Transactions on Mobile Computing*, 1(1), 2002.

[8] A. Woo and D. Culler. A Transmission Control Scheme for Media Access in Sensor Networks. In *Proc. 7th Ann. Intl. Conf. on Mobile Computing and Networking*, pages 221–235, Rome, Italy, July 2001. ACM.

[9] L. Charlie Zhong, R. C. Shah, C. Guo, and J. M. Rabaey. An Ultra-Low Power and 83 Distributed Access Protocol for Broadband Wireless Sensor Networks. In *IEEE Broadband Wireless Summit*, Las Vegas, NV, May 2001.

[10] R. German. Simulation and Modeling I Lecture Notes in Computer Science. *Erlangen-Nürnberg University, Erlangen*, Winter Term 2003/2004.

[11] *SCADDS: Scalable Coordination Architectures for Deeply Distributed Systems* web page.http://www.isi.edu/scadds/projects/smac/

[12] NO *Ad-Hoc Routing Agent (NOAH)* web page.
http://icapeople.epfl.ch/widmer/uwb/ns-2/noah/

[13] *The Network Simulator - ns-2* homepage.
http://www.isi.edu/nsnam/ns/

[14] The VINT project. The NS Manual. *UC Berkeley, LBL, USC/ISI, and Xerox PARC.*
http://www.isi.edu/nsnam/ns/doc/ns_doc.pdf

[15] M. Greis. Tutorial for the network simulator ns.
http://www.isi.edu/nsnam/ns/tutorial/index.html

[16] *The Network Simulator: Building Ns* web page.
http://www.isi.edu/nsnam/ns/ns-build.html

[17] *NsNam Site Search* web page.
http://www.isi.edu/nsnam/htdig/search.html

[18] J. Chung and M. Claypool. NS by example
Worcester Polytechnic Institute. The Department of Computer Science.
http://nile.wpi.edu/NS/

[19] *Smac-users -- Discussions by users of S-MAC* web page.
http://mailman.isi.edu/mailman/listinfo/smac-users

[20] *Energy Model Update in ns-2* web page.
http://www.isi.edu/ilense/software/smac/ns2_energy.html

[21] Awk tutorial,
http://www.vectorsite.net/tsawk.html.

[22] Gnuplot download and tutorial,
http://gnuplot.sourceforge.net/.