8 Algorithms for Wireless Sensor Networks

Talk is cheap. Show me the code.

—Linus Torvalds

An algorithm can be defined as a logical sequence of instructions for solving a problem in a finite sequence of steps. In wireless sensor networks, the design of algorithms becomes an important issue as energy and computational resources are scarce and therefore must be effectively put to good use. With the limited computational ability of each individual node, multiple sensors nodes collaborate to solve tasks using complex parallel processing techniques. These parallel processing techniques rely on efficient parallel algorithms to achieve collaboration. Therefore, in this context, a parallel algorithm can be defined as an algorithm in which several computations are carried on simultaneously across multiple processing units. In order to extend the life of a sensor network, these parallel algorithms have to be developed to be efficient in network resource usage; hence, the need for energy-aware networking algorithms [7,10,9]. There are numerous networked computing devices of all shapes and sizes from handheld computers such as personal digital assistants and mobile phones, to more powerful systems such as laptops, desktop workstations, and supercomputers. Most of these devices communicate over traditional IP-based networks supporting huge data transfer rates due to rapidly increasing bandwidth and faster processing capabilities. These networks, being highly scalable and structured, may consist of several routers, switches, and bridges interconnecting millions of nodes and may use complex routing schemes to transfer data from one end device to another. With energy and computational resource usage being of no consequence (energy can be replenished for most devices), communication between multiple devices is very cheap and reliable. It is in this respect that wireless sensor networks chiefly differ from conventional networks. With nonrenewable sources of power and very little on-board power, computation and communication come at a very high price. As wireless sensors become more pervasive because of their lower cost, we can expect the proliferation of sensors to exceed those of traditional computing devices. Hence, there is a need to develop new energy-aware routing algorithms and aggressive power
management schemes for this newly emergent class of computing devices [4]. In this chapter, we will discuss several concepts, challenges, properties, and algorithms unique to wireless sensor networks, such as

- Communication patterns prevalent in sensor networks
- Physical components of sensor nodes
- Properties of wireless sensor networks
- Networking layers
- Routing

Also, we will show how some of these concepts can be implemented using nesC code and pseudocode.

### 8.1 STRUCTURAL CHARACTERISTICS OF SENSOR NODES

The term *sensor nodes* (also called *motes*) refers to a sensing device that belongs to a wireless sensor network that is capable of processing, gathering, and communicating sensory information with other devices in the network. The architecture of a typical mote is shown in Fig. 8.1.

With advances in microelectromechanical systems (MEMSs) and low-power wireless technology, the major components of sensors as shown in Fig. 8.1 have been miniaturized over the years. However, the processing and storage capabilities of sensors have not developed in accordance with Moore’s law. In the following section, we compare the properties of some of these components (microcontroller, transceiver, memory, power unit, and sensors) with their traditional counterparts.

![Diagram of sensor nodes](image-url)

**FIGURE 8.1** Structure of sensor nodes.
8.1.1 Microcontroller

The microcontroller is one of the most important components in a sensor. It is responsible for data processing, memory management, interrupt handling, and the control of other components on the sensor to minimize energy usage and increase the lifespan of the motes. Unlike the microcontrollers present in traditional computers, most sensors operate on less than a watt of power, giving them a lifespan of several months to a year. This power conservation is achievable because of the ability of microcontrollers to enter low-power states when idling. These microcontrollers have frequencies between a few kilohertz and several megahertz, allowing for the most basic tasks such as sensing and intermediate routing of data from other sensors.

8.1.2 Transceiver

Currently, information can be transmitted in wireless sensor networks through three types of media:

- Optical communication (laser)
- Infrared (IR) communication
- Radio frequency (RF)

Although both optical and IR communications have the advantage of requiring less energy to transmit data, they both suffer from major drawbacks. For instance, optics-based communications require both communicating nodes to be properly aligned in direct line of sight. Conversely, IR communications have very short ranges. Other issues include sensitivity to atmospheric conditions in optical communications and unidirectional communication for both systems. It is for these reasons that RF is more prevalent in sensor networks.

8.1.3 Memory

As noted in earlier chapters, memory is a scarce resource on sensor nodes. Therefore, applications must be efficient in terms of not only energy but also the amount of memory they use. For instance, a mica2 mote contains 644 kb of memory that must be used by the TinyOS operating system and applications built on the platform [1].

8.1.4 Power Unit

Most power consumption in sensor nodes is primarily for data processing, sensing, or communication operations. Of these operations, communication activities exert the most toll on the battery life of the sensor. It is for this reason that new low-power components are being created and power-saving policies, such as dynamic power management (DPM) and dynamic voltage scaling (DVS), are continually being refined to provide ever-increasing energy savings. A typical sensor node is shown in Fig. 8.2.
8.2 DISTINCTIVE PROPERTIES OF WIRELESS SENSOR NETWORKS

With the unpredictable nature and unsuitability of traditional routing algorithms in wireless sensor networks, newer, more efficient, and more reliable algorithms have been developed. These algorithms were developed to address the challenges posed by the volatile wireless communication in sensor networks. These challenges stem from the ubiquitous and heterogeneous nature of WSNs, making central management of individual sensor nodes nearly impossible. As a result, intelligent features such as self-configuration, healing, dynamic routing, and multihop communication abilities have been suggested to improve the reliability and management tasks in these networks. In the following section, we discuss each of these attributes and how they enhance communication in wireless sensor networks.

8.2.1 Self-Configuration

Wireless sensor networks are usually composed of thousands of nodes randomly deployed and organized in order to achieve similar objectives (see Fig. 8.3 for an example). These objectives are to retrieve certain data about an entity being monitored and transmit results to a remote destination serving as a data sink. The random deployment of sensors and the volatile nature of the network (usually due to node failure) warrant the development of self-configuration mechanisms to prevent network degradation and efficient transmission of information.

8.2.2 Self-Healing

Despite measures taken to ensure durable sensor networks, several factors still exist that can result in a breakdown in communication. These factors include energy depletion in some key routing nodes, (un)intentional damage by humans or other animals, or even the addition of new nodes resulting in a highly dynamic network. It
8.2.3 Dynamic Routing

Communication in sensor networks can be very expensive, and sensor nodes can be added and removed on the fly. The need for an adaptive routing scheme based on network conditions such as link quality, hop count, and gradients has led to the development of several new on-demand routing algorithms. These make use of lower band width for control packets, resulting in less need to food the network for periodic updates. This, in turn, results in energy savings for sensor nodes, extending the life of a wireless sensor network considerably. Examples of dynamic routing in wireless sensor networks include caching and multipath (CHAMP) routing and hierarchical state routing (HSR).

We have now examined the structural characteristics of sensors and some of the unique properties of wireless sensor networks. In the following section we will discuss the components forming the network stack in wireless sensor networks.

8.3 SENSOR NETWORK STACK

Communication, routing, and data transfer in sensor networks is possible between differing node types because of well-established standards and specifications providing low-level implementation details on how data can be exchanged in sensor networks.
The IEEE 802.15.4 is the standard specifying details on data exchange in the physical and medium access control (MAC) layers for low-rate wireless networks. When sensor network–based applications are written, most interactions with the IEEE-specified layers are usually through abstract libraries implemented by independent vendors and offered through specific sensor operating systems (e.g., TinyOS). These libraries automatically transform application data into a form in conformance with IEEE specifications. These vendor-provided libraries providing software abstraction for the MAC and physical layers are referred to as the network layer. In this section, the several components that form the networking stack in wireless sensor networks are examined (see also Fig. 8.4).

### 8.3.1 Physical Layer

The physical layer is the first and lowest layer consisting of basic hardware transmission technologies of a network. It is responsible for several functions, including

- Provision of a data transmission service.
- Management of RF transceiver.
- Channel selection.
Energy and signal management functions. For wireless sensor networks, the physical layer transmits in one of three unlicensed frequency bands. In North America the most common is the 915-MHz ISM (instrument–scientific–medical) band. Among the main functions of the physical layer is the detection and correction of transmission errors. These errors could stem from several factors, such as

*Attenuation*—a decrease in intensity of electromagnetic energy at receiver due to long distance.

*Doppler shift*—a change in frequency of a wave caused by the relative velocities of the transmitter and receiver (common for mobile agents).

*Hidden-terminal problem*—a scenario in which the medium around the source node is free but busy around the destination node.

*Exposed-terminal problem*—in this case, the medium around the destination node is free but engaged at the source node.

### 8.3.2 Medium Access Control (MAC) Layer

The MAC layer is the second lowest layer, offering a management interface for the physical channel. It is responsible for frame validation, timeslot allocation, synchronization, and node associations in a network. There are several other implementations of the MAC Layer for sensor networks, such as S-MAC, B-MAC, and C-MAC, all of which have different strengths ranging from energy conservation to routing speed gains. References 8, 6, and 11 provide more information on MAC implementations in wireless sensor networks.

### 8.3.3 Network Layer

Although there is no defined standard for the network layer in sensor networks, several differing implementations exist today, the most common of which is the Zigbee specification. Details on the Zigbee network and application layer are covered in Chapter 10.

### 8.3.4 Full-Function Device (FFD)

Full-function devices are nodes having a general model of communication allowing them to “talk” with any other device. Also, such a device can be assigned the role of coordinator of a personal area network.

### 8.3.5 Reduced-Function Device (RFD)

Reduced-function devices are more restricted in their functions. They are usually very simple devices with low resource and communication requirements. It is for this reason that they can communicate only with FFDs and can never become network coordinators.
8.4 SYNCHRONIZATION IN WIRELESS SENSOR NETWORKS

*Time synchronization* refers to a method of timekeeping requiring the coordination of multiple events to operate a system in unison. It is particularly important in computing because of its critical role in communication. Distributed systems rely on several synchronization mechanisms to ensure proper operation without which severe degradation in performance can occur [5, 2, 3]. In data aggregation and event monitoring sensor networks, the need for a flexible and robust time synchronization mechanism is more apparent since collaboration among nodes is critical to data reduction and the energy efficiency of sensor networks. Also, sensor usually monitor highly dynamic environments that change with time. For this reason, time is a basic requirement for nodes to correlate events that occur in the network with events in the physical world. Failure to synchronize sensor nodes in a sensor network would render measurements useless since synchronization is essential for both temporal and spatial analysis of events (most tasks fall into these two categories, such as node localization, and target tracking). Several methods exist to synchronize nodes in a network, the most basic of which is sender–receiver or receiver–receiver synchronization. In this approach, peers in a network conduct time synchronization using timestamps in data packets as a reference. A visual representation of the algorithm and the accompanying equations are shown in Fig. 8.5.

Some drawbacks exist with this basic method of synchronization. Most notable are the four different types of delays involved:

- **Send time**—time spent in building the packet for transmission and delays inherent in the protocol stack
- **Access time**—time spent waiting for the medium to become free
• *Propagation time*—time taken for the packet to reach the destination
• *Receive time*—time spent by the receiver processing the packet before it is timestamped

Although send and access time delays can be eliminated with the use of reference broadcast synchronization, some newer approaches exist that are less error-prone.

We will now discuss the programming challenges involved in implementing a few basic algorithms [4] from the WSN area. First, the beaconing behavior of surrounding nodes is examined.

### 8.4.1 Beaconing

*Beaconing* refers to the continuous transmission of small control packets that notify neighboring nodes about the presence of the transmitter. The pseudocode describing how beaconing works is presented below.

```plaintext
begin:
  while (true):
    broadcast (address, random time);
    sleep (random time);
end
```

As we did before, we assume that the node has the three basic capabilities (sense, broadcast and sleep) as enumerated above. In this example, we use a procedural style of code

**Procedural Beaconing**

```plaintext
begin:
  senseMedium(signal):
    if(signal equals absent):
      return 'no_signal'
    elif(signal equals weak):
      return 'medium_busy_or_no_data'
    elif(signal equals collision):
      return 'collision'
    elif(signal equals strong):
      return 'strong'
end
```

Sometimes certain routing schemes may employ the use of lists to keep track of their nearest neighbors. A sample pseudocode and associated nesC configuration showing how a neighborhood table construction can be done is shown below.
ALGORITHMS FOR WIRELESS SENSOR NETWORKS

{ initialize-table(table); i = 0; t1 = time-now();
  while (i < n){
    data-packet = sense-data();
    address = address-of(data-packet);
    if not-present(address, table)
      {add(address, table); i ++
       if (time-now() > t1 + max-time) break;}
  }
}

#define MOTE_AM_ID 10
configuration algorithm2AppC {}
implementation
{
  components MainC;
  components RandomC;
  components LedsC;
  components new AMSenderC(MOTE_AM_ID);
  components new AMReceiverC(MOTE_AM_ID);
  ActiveMessageAddressC
  components ActiveMessageAddressC;
  components algorithm2C;
  components ActiveMessageC;
  components new TimerMilliC();
  algorithm2C.Boot->MainC. Boot;
  algorithm2C.Random->RandomC;
  algorithm2C.Leds->LedsC;
  algorithm2C.AMPacket->AMSenderC;
  algorithm2C.AMSend->AMSenderC;
  algorithm2C.AMA->ActiveMessageAddressC;
  algorithm2C.PacketAcknowledgements->AMSenderC;
  algorithm2C.Receive->AMReceiverC;
  algorithm2C.RadioControl->ActiveMessageC;
}

8.4.2 Neighborhood Table Construction

In this application we use the standard interfaces shown in the program above.
Wiring for Neighborhood Table Construction  The interfaces are implemented using the system-provided components shown above. The implementation details are discussed below.

implementation
{
    void sendSensorInformation (uint 16_t);
    void populateSensorTable (uint 16_t, uint 32_t);
    int NO_OF_MOTES = 30
    sensorInfo* sensor;
    am_addr_t sensoraddress = 0;
    uint32_t sensormac = 0x001EDD32;
    // Pre-determined mac address of current sensor
    am_group_t sensorgroup = 1;
    sensorData table [NO_OF_MOTES];
    int index = 0;
    event void Boot.booted( )
    {
        call RadioControl.start( );
    }
}

{
    event void RadioControl.startDone(error_t err)
    {
        call AMA.setAddress(sensorgroup, sensoraddress);
        call Timer0.startOneShot(6000);
    }

    event void Timer0.fired ( )
    {
        message_t msg;
        sensor = (sensor Info*) call
        AMSend. getPayload (&msg );
        sensor->id = sensoraddress ;
        call AMSend. send (AM_BROADCAST ADDR, &msg,
        sizeof (sensorData));
        call Leds.led0Toggle( );
    }
}

We will assume that there are no more than 30 motes in the neighborhood at any given time. The mote address of the given node (where we want to build the neighborhood table) is 0, its MAC address is 0x001EDD32, and the identity of the group (which consists of the current node and all its neighbors) is 1. We begin the table construction by starting the radio.
142 ALGORITHMS FOR WIRELESS SENSOR NETWORKS

Starting the Radio Component  We set the group address to 1 and the current node address to 0, and then start a monoshot clock that invokes the event function Timer0.fired(), where we broadcast our node address to all neighbors and wait for their replies.

8.4.3 Implementation

Implementation Details of the Receive Method  As soon as we receive a message from any neighbor, say P, we check whether it is a reply to the message that we sent, and if it is, we store P’s ID in our table. However, if the message is a request asking us to send our ID, we then send our ID to node P. To achieve these operations, we use the following functions: sendSensorInformation and populateSensorTable.

```c
void sendSensorInformation (uint16_t receiver)
{
    message_t msg;
    sensor = (sensorInfo*) call AMSend. getPayload (&msg);
    sensor->id = sensoraddress;
    sensor->macAddress = sensormac;
    call AMSend. send (receiver, &msg, sizeof(sensorData));
}
```

```c
event message_t* Receive.receive (message_t* msg, void*
    payload, uint8_t len)
{
    sensorInfo* myInfo; //Specified in header file
    myInfo = (sensorInfo*)payload;
    if (myInfo->requestCode == 0x1)
    {
        call Leds.led2Toggle(); //Toggle Led 3 to
        indicate info received
        populateSensorTable (myInfo->id , myInfo-
        >macAddress);
    }
    else if (myInfo->requestCode == 0x0) //Send my info to some
        other mote
    {
        sendSensorInformation (myInfo->id);
        call Leds.led1Toggle(); //Toggle Led 2 to indicate
        request received
    }
    return msg;
}
```
void populateSensorTable(uint16 id, uint32 mac)
{
    int i = 0;
    // Simple check if current mac already exists in table
    while (i <= index && table[i].macAddress != mac)
    {
        i ++;
    }
    if (table[i].macAddress == mac)
    {
        table[i].moteID = id;
    }
    else
    {
        table[++i].moteID = id;
        table[i].macAddress = mac;
    }
}

Implementation Details of sendSensorInformation and populateSensorTable
Additionally, we need to provide implementations for the events in our program, which are invoked when we call AMA.setAddress, AMSend.send, and RadioControl.stop. In sendDone, when we fail to send a message, we turn on an LED and then do nothing.

stopDone, sendDone, and Changed Implementations
In wireless sensor networks, concurrent transmission by multiple nodes could result in transmission errors due to packet corruption. An example of such a scenario would be the widely studied hidden-terminal problem. As a result of this, steps have to be taken to reduce or eliminate the occurrence of errors. The collision avoidance algorithm presented below suggests a way to avoid such errors.

event void AMSend.sendDone(message msg, error err)
{
    if (err == FAIL)
    {
        call Leds.led0Toggle();
    }
}
async event void AMA.changed()
{}
event void RadioControl.stopDone(error err)
{}
8.5 COLLISION AVOIDANCE: TOKEN-BASED APPROACH

8.5.1 Token-Based Approach

The approach can be described very briefly using the following four abstract steps.

1. Create a token and pass it to a node.
2. If a node has the token, it transmits the data that it wants to transmit.
3. The node then transmits the token to its neighbor.
4. Use this idea to count the number of nodes in the network.

The state diagram in Fig. 8.6 captures the abstract behavior of each node in the token-based approach for collision avoidance. We implement this algorithm by building a component for it, starting with its interface.

```
module CollisionAvoidance_tokenBased
{
    uses interface Boot;
    uses interface LowPowerListening;
    interface Timer<TMilli> as Timer0;
    uses uses interface Send;
    uses interface AMSend;
    uses interface Receive;
    uses interface PacketAcknowledgement;
}
```

**FIGURE 8.6** State diagram for token-based approach for collision avoidance.
Component Interface Definition  The following program shows the wiring called CollisionAvoidance tokenBased. We need to build our component boot for booting the component, LowPowerListening for minimizing power consumption, Timer0 to run a clock with millisecond precision, send to send packets (of messages), AMSend to send packets to intended destinations, Receive to receive packets from other nodes, and finally, PacketAcknowledgement to indicate whether the packet sent is to be acknowledged. For these interfaces, we choose the following implementations (as they are already provided in the TinyOS system).

configuration transmitAppC( )
{} implementation
{
    components MainC;
    components Collision as AppC;
    components CC2420ActiveMessageC;
    components AMSenderC( );
    components new AMReceiverC( );
    components Timer<TMilli> as Timer0;
    AppC. Boot->MainC;
    AppC. Packet->AMSenderC;
    AppC. Send->AMSenderC; //Wrong component??
    AppC. PackageAcknowledgement->AMSenderC;
    AppC. Receive->AMReceiverC;
    AppC. Timer0->Timer0;
    AppC. LowPowerListening->CC2420ActiveMessageC;
}

Pseudocode Implementation
{
    event void Boot.booted( )
    {
        //Start one shot timer of Timer0 using
        Timer0.startOneShot (t1)
        //where t1 is sometime point in future when the
        clock fires once.
    }
    event void Timer0.fired( )
    {
        configure the LowPower Listening cycle of mote
    }
    event void Receive.received( )
    {
        configure the LowPower Listening cycle of mote
        send acknowledgement to sender;
    }
transmitAppC  For low-power listening, we wire our components to the modules that implements the CC2420 radiochip functionalities. For simplicity, we provide the implementation of our component in pseudocode.

Transmit Application Configuration File  When the monoshot timer triggers have fired(), we put our component into low-power listening mode, and then wait for someone to send us a token. When we receive a token, we send back an acknowledgment, perform any activities that we want to perform (sending messages to any node, etc.), and once finished, we hand over the token to a neighboring node.

8.5.2 Schedule-Based Communication

In order to minimize collision and save energy, we need to follow a systematic way of transmitting and sensing behavior at each node. In this section, we write programs that follow a schedule so that the transmitter transmits only when the listener is listening, and the listener is listening only when the transmitter is transmitting. There are two ways a schedule can be embedded at each node: explicit and implicit. In the explicit embedding, the schedule can be described using a data structure such as a list implemented using an array. In the implicit embedding, the schedule is implied in the sequence of operations that the node executes. In this example, we follow the implicit style.

We can build in the schedule implicitly at each node in its program as follows. Let $A:(0,B)$; mean at time $t=0$, $A$ can transmit a packet to $B$. Thus, $[[A:(0,B);B :(10,C); C:(20,D);D :(30,nil);E :(40,A);F :(50,E)]]$ can be a schedule. Total period $T = 60$ time units assuming that $F:(50,E)$ takes 10 units of time. The schedule will repeat periodically with a period $T$ of 60 time units.

We further illustrate in this example how we can specify the behavior of the node abstractly using rules. The value of the timer, as specified by the time globally, schedules the operations so that at each node’s operations are executed as per the intended schedule.

{  
  A: $t = 0$ transmit data packet to B.  
  B: $t =10$ transmit data packet to C.  
  C: $t =20$ transmit data packet to D.  
  D: NULL.  
  E: $t =40$ transmit data packet to A.  
  F: $t =50$ transmit data packet to E.  
}
8.5.3 Pseudocode at Each Node

Let \( t \) stand for the timer value. (We can implement this by building a counter that goes through states triggered by a clock.)

\[
\text{module transmitC}
\{
\text{uses interface Timer<TMilli> as Timer0;}
\text{uses interface Send;}
\text{uses interface Receive;}
\text{uses interface AMSend;}
\text{uses interface Receive;}
\}
\text{configuration transmitAppC}
\{
\text{implementation}
\{
\text{components MainC;}
\text{components transmitC as AppC;}
\text{components ActiveMessageC;}
\text{components AMSenderC();}
\text{components Timer<TMilli> as Timer0;}
\text{AppC. Boot -> MainC;}
\text{AppC. Packet -> AMSenderC;}
\text{AppC.AMSend -> AMSenderC;}
\text{AppC. Send -> AMSenderC;}
\text{AppC. Receive -> AMReceiverC;}
\text{AppC. Timer0 -> Timer0;}
\}
\}
\]

Pseudocode Implementation

\[
\text{event void Boot.booted()}
\{
\text{mode = initialize();}
\text{c = 0;}
\text{Timer0.startPeriodic();}
\}
\]

\[
\text{event void Timer0.fired()}
\{
\text{if (mode = transmit & counter = 0)}
\{
\text{AMsend( to node B, data);}
\text{c = c+10;}
\}
148  ALGORITHMS FOR WIRELESS SENSOR NETWORKS

    if (mode == transmit & counter = = 10)
    {
        AMsend(to node C, data);
        c = c +10;
    }
}

    event void Receive( )
    {
        if (mode==receive)
            data = packet;
    }

States of Transmit  The concept described above can be used to avoid collisions, as we discuss below:

- Divide time into slots.
- Insert permitted transitions at each slot. For example, the time axis is divided into intervals of period $T$. Each period $T$ is divided into $n$ slots. Thus, $T = [\text{slot 1, slot 2, \ldots, slot } n]$.
- Indicate the possible transmissions in each slot. For example, [slot 1: $A \rightarrow B$, $C \rightarrow D$; slot 2: $E \rightarrow F$; slot 3: $G \rightarrow H$, $I \rightarrow J$; etc.].
- Assign to each node.

The list of interfaces we used in developing this application component are shown in the program above, along with the components that we selected for their implementation.

Schedule-Based Transmission  In order to implement the schedule, we use a small counter that tells us when to transmit a packet. This counter is incremented every time a periodic clock fires. (Assume that clocks are synchronized for this implementation.)

8.6  CARRIER SENSING VERSUS DECODING

In Fig. 8.7, the green circle shows the decode range and the yellow circle shows the carrier sense. (Note that these circles are part of the manufacturer’s specs.) To decode, the signal must be strong; that is, the signal-to-noise ratio (SNR) must be high. To carrier sense, the signal can be weak; that is, SNR can below.

We will now write programs for nodes $B$, $C$, $D$, $E$, $F$ and measure the relative signal strengths. Verify with the manufacturer’s specs—that is yellow circle and green circle.
CARRIER SENSING VERSUS DECODING

8.6.1 RTS/CTS Handshake

Ready-to-send (RTS) and clear-to-send (CTS) messages are broadcast control bytes that are exchanged between the transmitter and the receiver to coordinate the transmission of data between them. A typical handshake protocol that uses this is shown in Fig. 8.8. In this example, we also show how to convert a timing diagram into programs. Node \( B \) wants to transmit a data packet to node \( C \). (Also worthy of note is the fact that RTS and CTS will have the MAC addresses of the sender and the receiver. Further, RTS, CTS, and the data packets will all have the duration \( T \) that it will take for the complete transaction. Similarly, a frame will indicate whether the transmission is unicast or broadcast.) The handshake protocol proceeds as follows:

1. Node \( B \) senses the medium, and goes to the next step when the medium is free.
2. Node \( B \) broadcasts an RTS. (Note that it is a broadcast, and not a unicast. Also, RTS contains the MAC addresses of \( B \) and \( C \).)
3. Nodes \( C \) and \( A \) hear it. The RTS contains the total duration \( d_1 \) up to \( B \) receiving an acknowledgment.
4. Node C realizes that the RTS is for itself, senses the channel, acquires it, and broadcasts a CTS. This also contains the duration $d_2$. Concurrently, when A hears the RTS, it realizes that it is not for itself, but it will set the NAV flag on (to indicate that the medium is busy) for the duration $d_1$.

5. When the CTS from C reaches B, B unicasts one packet of data to C. (*Note:* This unicast frame will contain the sender's and receiver's addresses, and also a flat stating that the frame is a unicast frame.) Concurrently, when D hears the CTS broadcast from C, it will set its NAV flag on for the duration $d_2$.

6. Node C receives the data from B, and then sends a unicast acknowledgment to B.

We now give the pseudocode for the protocol above.

**Node B:**
```
begin:
    //Broadcast RTS to C: First Acquire Medium
    while (true): //Infinite loop
        if (senseMedium( ) not 'BUSY'):
            break
        endIf
    endLoop
    //Listen for CTS from C
    while (true):
        packet <- listen( )
        if (packet equals 'CTS'):
            break
        endIf
    endLoop
```
CARRIER SENSING VERSUS DECODING

//Send Data to C
while (true):
    if (senseMedium( ) not 'BUSY'):
        break;
    endIf
endLoop
destination <-| C
unicast(data, destination)

//Wait for Acknowledgement from C
while (true):
    packet <-| listen( )
    if (packet equals 'CTS')
        break
    endIf
endLoop

Node C:
begin:

//Listen for RTS from B
while (true):
    packet <-| listen( )
    if (packet equals 'RTS'):
        break
    endIf
endLoop

//Broadcast CTS to B
while (true):
    if (senseMedium( ) not 'BUSY'):
        break;
    endIf
endLoop

//CTS Broadcast to B
while (true):
    packet <-| listen( )
    if (packet equals 'DATA'):
        break
    endIf
endLoop
print '"Data_received_from B"

//Broadcast ACK to B
while (true):
    if (senseMedium( ) not 'BUSY'):
        break
endLoop

Node A:
begin:
    while (true):
        packet <- listen( )
        if (packet equals 'RTS'):
            break
endLoop

//B is going to send data to C, Must send A to sleep
//NVA Variable
duration <- extractDuration (packet )
NVA = currentTime + duration
sleepUntil (NVA)
print "Woken up after sleeping!"

end

Node D:
begin:
    while (true):
        packet <- listen( )
        if (packet equals 'RTS'):
            break
endLoop

//B is going to send data to C, Must send D to sleep
//NVA Variable
duration <- extractDuration (packet )
NVA = currentTime + duration
sleepUntil (NVA)
print "Woken up after sleeping!"

end

Broadcast RTS  To keep the illustration simple, we have provided pseudocode for only one role at each node. The nesC pseudocode is given below.
**Theme: Simulation of RTS/CTS Handshake** In this chapter, we have illustrated several protocols and shown how they can be coded in the nesC language. We observed that coding the protocols themselves was fairly straightforward.

Node B:
```nesC
booted():
    send (RTS to C);
receive():
    if (CTS is received)
        transmit data;
    if (ACK is received)
        do nothing;
```

Node C
```nesC
booted()
{NIL}
receive():
    if (RTS received)
        send (CTS);
    if (data received)
        store (data);
    if (end of data received)
        send (ACK);
```

Node A
```nesC
booted():
{NIL}
receive():
    if (RTS received) then
        extract duration d1 from the packet;
    lowPowerSleep(for duration d1);
    // that is, set NAV = busy;
```

Node D
```nesC
booted():
{NIL}
receive():
    if (CTS is received) then
        extract duration d2 from the packet;
    lowPowerSleep(for duration d2);
    // that is, set NAV = busy;
```

**PROBLEMS**

8.1 How does the S-MAC differ from traditional wireless MAC?

8.2 What are the modes of operation of S-MAC?
8.3 What is idle listening?
8.4 What are the sources of energy drain in a sensor node?
8.5 What is synchronization? Show an implementation in which five nodes synchronize with each other.
8.6 Describe the major contents of a routing table and give an example of nodes in a network slowly building their routing neighborhood routing table.
8.7 Provide an example where data are routed using the previous routing table and describe how a routing table changes over time.
8.8 Describe the CSMA-CA mechanism.
8.9 Describe the TDMA mechanism and its advantages and disadvantages for sensor network applications.

REFERENCES