



ANNAI WOMEN'S COLLEGE

(Arts & Science)

(Affiliated to Bharathidasan University, Tiruchirappalli – 620 024)

Aurobindo Nagar, TNPL Road, Punnamchatram, Karur – 639 136.



Course Material

Paper Name : Wireless Sensor Networks

Paper Code : P16CS42

Subject Handler : S.Leelavathi

No.Of Units : 5 units

UNIT-I

OVERVIEW OF WIRELESS SENSOR NETWORKS & ARCHITECTURES

1.1 KEY DEFINITIONS OF SENSOR NETWORKS:

Definition: A Sensor Network is composed of a large number of sensor nodes, which are tightly positioned either inside the phenomenon or very close to it.

Sensor networks have the contribution from signal processing, networking and protocols, databases and information management, distributed algorithms, and embedded systems and architecture.

A wireless sensor network (WSN) can be defined as a network of low-size and lowcomplex devices denoted as nodes that can sense the environment and communicate the information gathered from the monitored field through wireless links.

The following are the Key terms and concepts that will be used in sensor network development techniques.

- Sensor: A transducer that converts a physical phenomenon such as heat, light, sound, or motion into electrical or other signals that may be further operated by other apparatus.
- Sensor node: A basic unit in a sensor network, with on-board sensors, processor, memory, wireless modem, and power supply. It is often abbreviated as *node*. When a node has only a single sensor on board, the node is sometimes referred as a *sensor*.
- Network topology: A connectivity graph where nodes are sensor nodes and edges are communication links. In a wireless network, the link represents a one-hop connection, and the neighbors of a node are those within the radio range of the node.
- Routing: The process of determining a network path from a packet source node to its destination.
- Date-centric: Approaches that name, route, or access a piece of data via properties, such as physical location, that are external to a communication network. This is to be contrasted with addresscentric approaches which use logical properties of nodes related to the network structure.
- Geographic routing: Routing of data based on geographical features such as locations or regions. This is an example of datecentric networking.
- In-network: A style of processing in which the data is processed and combined near where the data is generated.

- *Collaborative processing*: Sensors cooperatively processing data from multiple sources in order to serve a high-level task. This typically requires communication among a set of nodes.
- *State*: A snapshot about a physical environment (e.g., the number of signal sources, their locations or spatial extent, speed of movement), or a snapshot of the system itself (e.g., the network state).
- *Uncertainty*: A condition of the information caused by noise in sensor measurements, or lack of knowledge in models. The uncertainty affects the system's ability to estimate the state accurately and must be carefully modeled. Because of the ubiquity of uncertainty in the data, many sensor network estimation problems are cast in a statistical framework. For example, one may use a covariance matrix to characterize the uncertainty in a Gaussian-like process or more general probability distributions for non-Gaussian processes.
- *Task*: Either high-level system tasks which may include sensing, communication, processing, and resource allocation, or application tasks which may include detection, classification, localization, or tracking.
- *Detection*: The process of discovering the existence of a physical phenomenon. A threshold-based detector may flag a detection whenever the signature of a physical phenomenon is determined to be significant enough compared with the threshold.
- *Classification*: The assignment of class labels to a set of physical phenomena being observed.
- *Localization and tracking*: The estimation of the state of a physical entity such as a physical phenomenon or a sensor node from a set of measurements. Tracking produces a series of estimates over time.
- *Value of information or information utility*: A mapping of data to a scalar number, in the context of the overall system task and knowledge. For example, information utility of a piece of sensor data may be characterized by its relevance to an estimation task at hand and computed by a mutual information function.
- *Resource*: Resources include sensors, communication links, processors, on-board memory, and node energy reserves. Resource allocation assigns resources to tasks, typically optimizing some performance objective.
- *Sensor tasking*: The assignment of sensors to a particular task and the control of sensor state (e.g., on/off, pan/tilt) for accomplishing the task.
- *Node services*: Services such as time synchronization and node localization that enable applications to discover properties of a node and the nodes to organize themselves into a useful network.
- *Data storage*: Sensor information is stored, indexed, and accessed by applications. Storage may be local to the node where the data is generated, load-balanced across a network, or anchored at a few points (warehouses).
- *Embedded operating system (OS)*: The run-time system support for sensor network applications. An embedded OS typically provides an abstraction of system resources and a set of utilities.

- System performance goal: The abstract characterization of system properties. Examples include scalability, robustness, and network longevity, each of which may be measured by a set of evaluation metrics.
- Evaluation metric: A measurable quantity that describes how well the system is performing on some absolute scale. Examples include packet loss (system), network dwell time (system), track loss (application), false alarm rate (application), probability of correct association (application), location error (application), or processing latency (application/system). An evaluation method is a process for comparing the value of applying the metrics on an experimental system with that of some other benchmark system.

1.2 ADVANTAGES OF SENSOR NETWORKS:

Networked sensing offers unique advantages over traditional centralized approaches. Dense/ compressed networks of distributed communicating sensors can improve signal-to-noise ratio (SNR) by reducing average distances from sensor to source of signal, or target. Increased energy efficiency in communications is enabled by the multi-hop topology of the network. A decentralized sensing system is inherently more strong against individual sensor node or link failures, because of redundancy in the network.

1.2.1 Energy Advantage:

Because of the unique attenuation characteristics of radio-frequency (RF) signals, a multi-hop RF network provides a significant energy saving over a single-hop network for the same distance. Consider the following simple example of an N -hop network. Assume the overall distance for transmission is Nr , where r is the one-hop distance. The minimum receiving power at a node for a given transmission error rate is $P_{receive}$, and the power at a transmission node is P_{send} . Then, the RF attenuation model near the ground is given by $\frac{P_{receive}}{P_{send}} \propto \frac{1}{r^\alpha}$, where r is the transmission distance and α is the RF attenuation exponent.

Due to multipath and other interference effects, α is typically in the range of 2 to 5. Equivalently, $P_{send} \propto P_{receive} r^\alpha$.

Therefore, the power advantage of an N -hop transmission versus a single-hop transmission

$$\text{over the same distance } Nr \text{ is } \eta_{rf} = \frac{P_{send}(Nr)}{N \cdot P_{send}(r)} = \frac{(Nr)^\alpha P_{receive}}{N \cdot r^\alpha P_{receive}} = N^{\alpha-1} \quad \text{-----}$$

------(1)

Figure 1.1 illustrates the power attenuation for the multi-hop and single-hop networks.

A larger N gives a larger power saving due to the consideration of RF energy alone.

However, this analysis ignores the power usage by other components of an RF circuitry.

Using more nodes increases not only the cost, but also the power consumption of these other RF components. In practice, an optimal design seeks to balance the two conflicting factors for an

overall cost and energy efficiency. Latency and robustness considerations may also argue against an unduly large number of relay nodes.

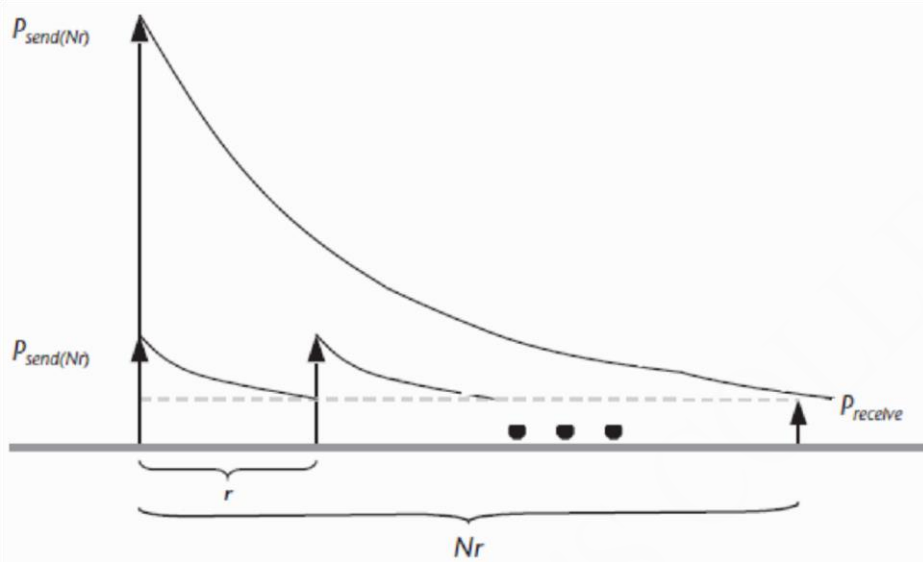


Figure 1.1: The power advantage of using a multi-hop RF communication over a distance of Nr

1.2.2 Detection Advantage:

Each sensor has a finite sensing range, determined by the noise floor at the sensor. A denser sensor field improves the odds of detecting a signal source within the range. Once a signal source is inside the sensing range of a sensor, further increasing the sensor density decreases the average distance from a sensor to the signal source, hence

improving the signal-to-noise ratio (SNR). Let us consider the acoustic sensing case in a plane, where the acoustic power P_{source} is received at a distance r is, which assumes an inverse attenuation. The SNR is given by

$$P_{receive} \propto \frac{P_{source}}{r^2}$$

two-dimensional distance squared

$$SNR_r = \frac{P_{receive}}{P_{Noise}} = \frac{P_{source}}{10 \log 20 \log r \cdot P_{Noise}} \quad \text{----- (2)}$$

Increasing the sensor density by a factor of k reduces the average distance to a target by a factor of $\frac{1}{\sqrt{k}}$. Thus, the SNR advantage of the denser sensor network is

$$20 \log \frac{r}{\frac{r}{\sqrt{k}}} = 10 \log k \quad \text{-----}$$

Hence, an increase in sensor density by a factor of k improves the SNR at a sensor by $10 \log k$ db.

1.3 UNIQUE CONSTRAINTS AND CHALLENGES:

1.3.1 Constraints: A sensor network has a unique set of resource constraints problems such as finite on-board battery power and limited network communication bandwidth. A sensor network consists of circulated self-governing sensors to monitor physical or environmental conditions. WSN consist of an array of sensors, each sensor network node has typically several parts such as radio, transceiver, antenna and microcontroller. A Base station links the sensor network to another network to advertise the data sensed for future processing. Each sensor node communicates wirelessly with a few other local nodes within its radio communication range. Sensor networks extend the existing Internet deep into the physical environment.

One of the biggest Constraint/problem of sensor network is power consumption. To solve this issue two methods are defined. First method is to introduce aggregation points (An aggregation is a collection, or the gathering of things together). This reduces total number of messages exchanged between nodes and saves some energy. Usually aggregation points are ordinary nodes that receive data from neighbouring nodes, execute processing and then forward the filtered data to next hop.

Real-time is a very important constraint in WSNs, because real-world conditions can introduce explicit or implicit time constraints. These networks are supposed to sense signals in the environment, and concepts like “data freshness” are important in its applications. This way, in some application, time-based/temporal validity in data collect by nodes can expire very quickly.

1.3.2 Challenges: The challenges we face in designing sensor network systems and applications include Limited hardware, Limited support for networking, Limited support for software development.

- *Limited hardware:* Each node has limited processing, storage, and communication capabilities, and limited energy supply and bandwidth.
- *Limited support for networking:* The network is peer-to-peer, with a mesh topology and dynamic, mobile, and unreliable connectivity. There are no universal routing protocols or central registry services. Each node acts both as a router and as an application host.
- *Limited support for software development:* The tasks are typically real-time and massively distributed, involve dynamic teamwork among nodes, and must handle multiple competing events. Global properties can be specified only via local instructions. Because of the coupling between applications and system layers, the software architecture must be codesigned with the information processing architecture

1.4. DRIVING APPLICATIONS:

Sensor networks may consist of many different types of sensors such as magnetic, thermal, visual, seismic, infrared and radar, which are able to monitor a wide variety of conditions. These sensor nodes can be put for continuous sensing, location sensing, motion sensing and event detection. The idea of micro-sensing and wireless connection of these sensor nodes promises many new application areas. A few examples of their applications are as follows:

A. Area monitoring applications

Area monitoring is a very common application of WSNs. In area monitoring, the WSN is deployed over a region where some physical activity or phenomenon is to be monitored. When the sensors detect the event being monitored (sound, vibration), the event is reported to the base station, which then takes appropriate action (e.g., send a message on the internet or to a satellite). Similarly, wireless sensor networks can be deployed in security systems to detect motion of the unwanted, traffic control system to detect the presence of high-speed vehicles. Also WSNs finds huge application in military area for battleeld surveillance, monitoring friendly forces, equipment and ammunition, reconnaissance of opposing forces and terrain, targeting and battle damage assessment .

B. Environmental applications

A few environmental applications of sensor networks include forest fire detection, green house monitoring, landslide detection, air pollution detection and flood detection. They can also be used for tracking the movement of insects, birds and small animals, planetary exploration, monitoring conditions that affect crops and livestock and facilitating irrigation.

C. Health applications

Some of the health applications for sensor networks are providing interfaces for the disabled, integrated patient monitoring, diagnostics, drug administration in hospitals, monitoring the movements and internal processes of insects or other small animals, telemonitoring of human physiological data, and tracking and monitoring doctors and patients inside a hospital.

D. Industrial applications

WSNs are now widely used in industries, for example in machinery condition-based maintenance. Previously inaccessible locations, rotating machinery, hazardous or

restricted areas, and mobile assets can now be reached with wireless sensors. They can also be used to measure and monitor the water levels within all ground wells and monitor leachate accumulation and removal.

E. Other applications

Sensor networks now find huge application in our day-to-day appliances like vacuum cleaners, micro-wave ovens, VCRs and refrigerators. Other commercial applications includes constructing smart ope spaces, monitoring product quality, managing inventory, factory instrumentation and many more.

1.5 ENABLING TECHNOLOGIES FOR WIRELESS SENSOR NETWORKS:

Building such wireless sensor networks has only become possible with some fundamental advances in enabling technologies.

First technology is the miniaturization of hardware. Smaller feature sizes in chips have driven down the power consumption of the basic components of a sensor node to a level that the constructions of WSNs can be planned. This is particularly relevant to microcontrollers and memory chips and the radio modems which are responsible for wireless communication have become much more energy efficient. Reduced chip size and improved energy efficiency is accompanied by reduced cost.

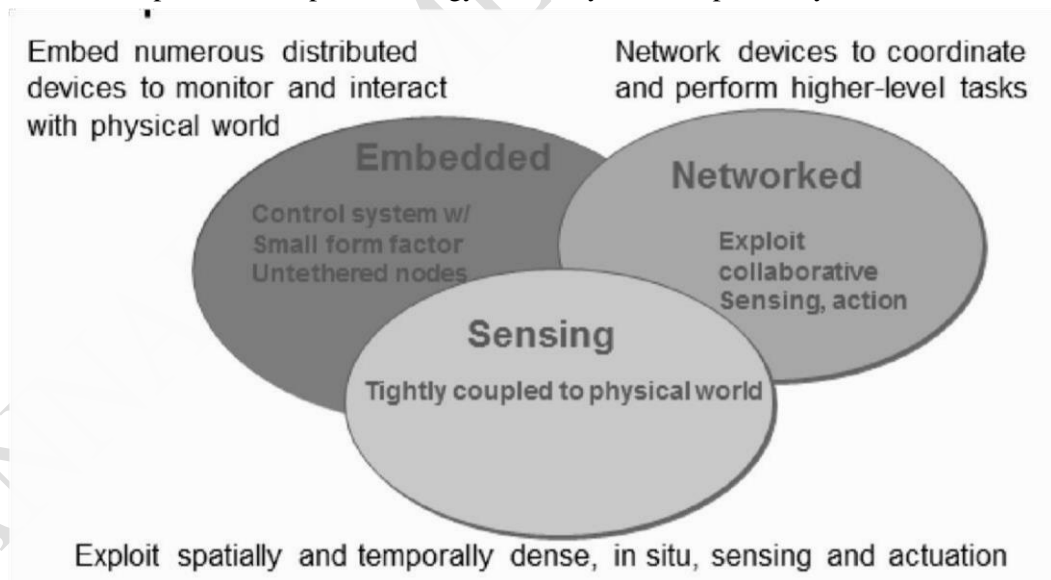


Figure 1.2: Enabling Technologies

Second one is processing and communication and the actual sensing equipment is the third relevant technology. Here, however, it is difficult to generalize because of the vast range of possible sensors.

These three basic parts of a sensor node have to be accompanied by power supply. This requires, depending on application, high capacity batteries that last for long times, that is, have only a negligible self-discharge rate, and that can efficiently provide small amounts of current. Ideally, a sensor node also has a device for **energy scavenging**, recharging the battery with energy gathered from the environment – solar cells or vibration-based power generation are conceivable options. Such a concept requires the battery to be efficiently chargeable with small amounts of current, which is not a standard ability. Both batteries and energy scavenging are still objects of ongoing research.

The counterpart to the basic hardware technologies is software. This software architecture on a single node has to be extended to a network architecture, where the division of tasks between nodes, not only on a single node, becomes the relevant question – for example, how to structure interfaces for application programmers. The third part to solve then is the question of how to design appropriate communication protocols.

SINGLE-NODE ARCHITECTURE:

1.6 HARDWARE COMPONENTS: Choosing the hardware components for a wireless sensor node, obviously the applications has to consider size, costs, and energy consumption of the nodes. A basic sensor node comprises five main components such as Controller, Memory, Sensors and Actuators, Communication devices and Power supply Unit.

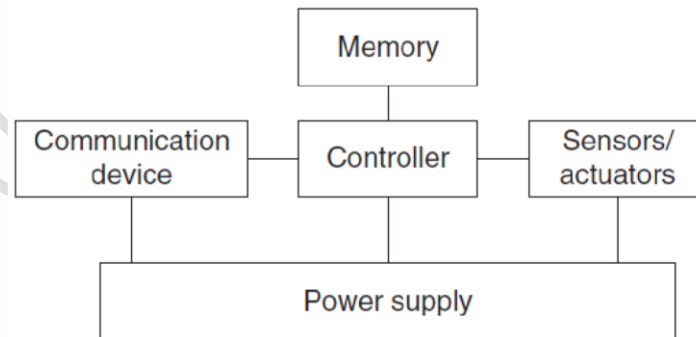


Figure 1.3: Sensor node Hardware components

1.6.1 Controller: A controller to process all the relevant data, capable of executing arbitrary code. The controller is the core of a wireless sensor node. It collects data from the sensors, processes this data, decides when and where to send it, receives data from other sensor nodes, and decides on the actuator's behavior. It has to execute various

programs, ranging from timecritical signal processing and communication protocols to application programs; it is the Central Processing Unit (CPU) of the node.

For General-purpose processors applications microcontrollers are used. These are highly overpowered, and their energy consumption is excessive. These are used in embedded systems. Some of the key characteristics of microcontrollers are particularly suited to embedded systems are their flexibility in connecting with other devices like sensors and they are also convenient in that they often have memory built in.

A specialized case of programmable processors are Digital Signal Processors (DSPs). They are specifically geared, with respect to their architecture and their instruction set, for processing large amounts of vectorial data, as is typically the case in signal processing applications. In a wireless sensor node, such a DSP could be used to process data coming from a simple analog, wireless communication device to extract a digital data stream. In broadband wireless communication, DSPs are an appropriate and successfully used platform.

An FPGA can be reprogrammed (or rather reconfigured) “in the field” to adapt to a changing set of requirements; however, this can take time and energy – it is not practical to reprogram an FPGA at the same frequency as a microcontroller could change between different programs.

An ASIC is a specialized processor, custom designed for a given application such as, for example, high-speed routers and switches. The typical trade-off here is loss of flexibility in return for a considerably better energy efficiency and performance. On the other hand, where a microcontroller requires software development, ASICs provide the same functionality in hardware, resulting in potentially more costly hardware development.

Examples: Intel Strong ARM, Texas Instruments MSP 430, Atmel ATmega.

1.6.2 Memory: Some memory to store programs and intermediate data; usually, different types of memory are used for programs and data. In WSN there is a need for Random Access Memory (RAM) to store intermediate sensor readings, packets from other nodes, and so on. While RAM is fast, its main disadvantage is that it loses its content if power supply is interrupted. Program code can be stored in Read-Only Memory (ROM) or, more typically, in Electrically Erasable Programmable Read-Only Memory (EEPROM) or flash memory (the later being similar to EEPROM but allowing data to be erased or written in blocks instead of only a byte at a time). Flash memory can also serve as intermediate storage of data in case RAM is insufficient or when the power supply of RAM should be shut down for some time.

1.6.3 Communication Device: Turning nodes into a network requires a device for sending and receiving information over a wireless channel.

Choice of transmission medium: The communication device is used to exchange data between individual nodes. In some cases, wired communication can actually be the method of choice and is frequently applied in many sensor networks. The case of wireless communication is considerably more interesting because it includes radio frequencies. Radio Frequency (RF) based communication is by far the most relevant one as it best fits the requirements of most WSN applications.

Transceivers: For communication, both transmitter and receiver are required in a sensor node to convert a bit stream coming from a microcontroller and convert them to and from radio waves. For two tasks a combined device called transceiver is used.

Transceiver structure has two parts as Radio Frequency (RF) front end and the baseband part.

1. The radio frequency front end performs analog signal processing in the actual radio frequency Band.
2. The baseband processor performs all signal processing in the digital domain and communicates with a sensor node's processor or other digital circuitry.

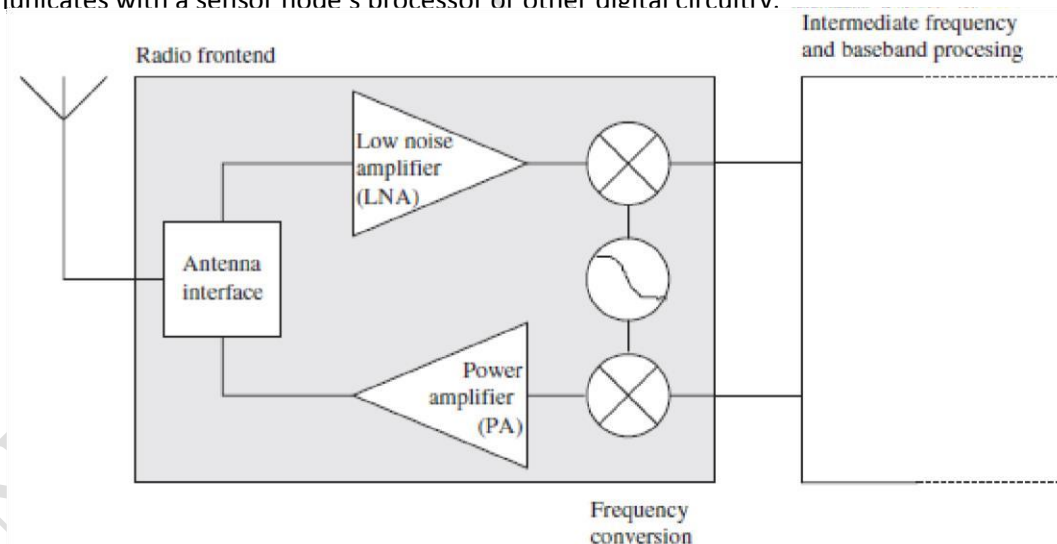


Figure 1. 4: RF front end

- ✓ The Power Amplifier (PA) accepts upconverted signals from the IF or baseband part and amplifies them for transmission over the antenna.
- ✓ The Low Noise Amplifier (LNA) amplifies incoming signals up to levels suitable for further processing without significantly reducing the SNR. The range of powers of the incoming signals varies from very weak signals from nodes close to the reception boundary to strong signals from nearby nodes; this range can be up to 100 dB.

- ✓ Elements like local oscillators or voltage-controlled oscillators and mixers are used for frequency conversion from the RF spectrum to intermediate frequencies or to the baseband. The incoming signal at RF frequencies f_{RF} is multiplied in a mixer with a fixed frequency signal from the local oscillator (frequency f_{LO}). The resulting intermediate frequency signal has frequency $f_{LO} - f_{RF}$. Depending on the RF front end architecture, other elements like filters are also present.

Transceiver tasks and characteristics:

- ☞ *Service to upper layer:* A receiver has to offer certain services to the upper layers, most notably to the Medium Access Control (MAC) layer. Sometimes, this service is packet oriented; sometimes, a transceiver only provides a byte interface or even only a bit interface to the microcontroller.
- ☞ *Power consumption and energy efficiency:* The simplest interpretation of energy efficiency is the energy required to transmit and receive a single bit.
- ☞ *Carrier frequency and multiple channels:* Transceivers are available for different carrier frequencies; evidently, it must match application requirements and regulatory restrictions.
- ☞ *State change times and energy:* A transceiver can operate in different modes: sending or receiving, use different channels, or be in different power-safe states.
- ☞ *Data rates:* Carrier frequency and used bandwidth together with modulation and coding determine the gross data rate.
- ☞ *Modulations:* The transceivers typically support one or several of on/off-keying, ASK, FSK, or similar modulations.
- ☞ *Coding:* Some transceivers allow various coding schemes to be selected.
- ☞ *Transmission power control:* Some transceivers can directly provide control over the transmission power to be used; some require some external circuitry for that purpose. Usually, only a discrete number of power levels are available from which the actual transmission power can be chosen. Maximum output power is usually determined by regulations.
- ☞ *Noise figure:* The noise figure NF of an element is defined as the ratio of the Signal-to-Noise Ratio (SNR) ratio SNR_I at the input of the element to the SNR ratio SNR_O at the element's output: $NF = \frac{SNR_I}{SNR_O}$. It describes the degradation of SNR due to the element's operation and is typically given in dB: $NF \text{ dB} = SNR_I \text{ dB} - SNR_O \text{ dB}$.
- ☞ *Gain:* The gain is the ratio of the output signal power to the input signal power and is typically given in dB. Amplifiers with high gain are desirable to achieve good energy efficiency.
- ☞ *Power efficiency:* The efficiency of the radio front end is given as the ratio of the radiated power to the overall power consumed by the front end; for a power

amplifier, the efficiency describes the ratio of the output signal's power to the power consumed by the overall power amplifier.

- ☞ *Receiver sensitivity*: The receiver sensitivity (given in dBm) specifies the minimum signal power at the receiver needed to achieve a prescribed E_b/N_0 or a prescribed bit/packet error rate.
- ☞ *Range*: The range of a transmitter is clear. The range is considered in absence of interference; it evidently depends on the maximum transmission power, on the antenna characteristics.
- ☞ *Blocking performance*: The blocking performance of a receiver is its achieved bit error rate in the presence of an interferer.
- ☞ *Out of band emission*: The inverse to adjacent channel suppression is the out of band emission of a transmitter. To limit disturbance of other systems, or of the WSN itself in a multichannel setup, the transmitter should produce as little as possible of transmission power outside of its prescribed bandwidth, centered around the carrier frequency.
- ☞ *Carrier sense and RSSI*: In many medium access control protocols, sensing whether the wireless channel, the carrier, is busy (another node is transmitting) is a critical information. The receiver has to be able to provide that information. The signal strength at which an incoming data packet has been received can provide useful information a receiver has to provide this information in the Received Signal Strength Indicator (RSSI).
- ☞ *Frequency stability*: The frequency stability denotes the degree of variation from nominal center frequencies when environmental conditions of oscillators like temperature or pressure change.
- ☞ *Voltage range*: Transceivers should operate reliably over a range of supply voltages. Otherwise, inefficient voltage stabilization circuitry is required.

1.6.4 Sensors and actuators: The actual interface to the physical world: devices that can observe or control physical parameters of the environment. **Sensors** can be roughly categorized into three categories as

- ☞ *Passive, omnidirectional sensors*: These sensors can measure a physical quantity at the point of the sensor node without actually manipulating the environment by active probing – in this sense, they are passive. Moreover, some of these sensors actually are self-powered in the sense that they obtain the energy they need from the environment – energy is only needed to amplify their analog signal.
- ☞ **Passive, narrow-beam sensors** These sensors are passive as well, but have a welldefined notion of direction of measurement.
- ☞ **Active sensors** This last group of sensors actively probes the environment, for example, a sonar or radar sensor or some types of seismic sensors, which generate shock waves by small explosions. These are quite specific – triggering an explosion is certainly not a lightly undertaken action – and require quite special attention.

Actuators: Actuators are just about as diverse as sensors, yet for the purposes of designing a WSN that converts electrical signals into physical phenomenon.

1.6.5 Power supply: As usually no tethered power supply is available, some form of batteries are necessary to provide energy. Sometimes, some form of recharging by obtaining energy from the environment is available as well (e.g. solar cells). There are essentially two aspects: Storing energy and Energy scavenging.

Storing energy: Batteries

- ☞ **Traditional batteries:** The power source of a sensor node is a battery, either nonrechargeable (“primary batteries”) or, if an energy scavenging device is present on the node, also rechargeable (“secondary batteries”).

Primary batteries			
Chemistry	Zinc-air	Lithium	Alkaline
Energy (J/cm ³)	3780	2880	1200
Secondary batteries			
Chemistry	Lithium	NiMHd	NiCd
Energy (J/cm ³)	1080	860	650

TABLE 1.1: Energy densities for various primary and secondary battery types

Upon these batteries the requirements are

- ☞ **Capacity:** They should have high capacity at a small weight, small volume, and low price. The main metric is energy per volume, J/cm³.
- ☞ **Capacity under load:** They should withstand various usage patterns as a sensor node can consume quite different levels of power over time and actually draw high current in certain operation modes.
- ☞ **Self-discharge:** Their self-discharge should be low. Zinc-air batteries, for example, have only a very short lifetime (on the order of weeks).
- ☞ **Efficient recharging:** Recharging should be efficient even at low and intermittently available recharge power.
- ☞ **Relaxation:** Their relaxation effect – the seeming self-recharging of an empty or almost empty battery when no current is drawn from it, based on chemical diffusion processes within the cell – should be clearly understood. Battery lifetime and usable capacity is considerably extended if this effect is leveraged.
- ☞ **DC–DC Conversion:** Unfortunately, batteries alone are not sufficient as a direct power source for a sensor node. One typical problem is the reduction of a battery’s voltage as its capacity drops. A DC – DC converter can be used to overcome this

problem by regulating the voltage delivered to the node's circuitry. To ensure a constant voltage even though the battery's supply voltage drops, the DC – DC converter has to draw increasingly higher current from the battery when the battery is already becoming weak, speeding up battery death. The DC – DC converter does consume energy for its own operation, reducing overall efficiency.

Energy scavenging: Depending on application, high capacity batteries that last for long times, that is, have only a negligible self-discharge rate, and that can efficiently provide small amounts of current. Ideally, a sensor node also has a device for **energy scavenging**, recharging the battery with energy gathered from the environment – solar cells or vibration-based power generation are conceivable options.

- ☞ *Photovoltaics*: The well-known solar cells can be used to power sensor nodes. The available power depends on whether nodes are used outdoors or indoors, and on time of day and whether for outdoor usage. The resulting power is somewhere between $10 \mu\text{W}/\text{cm}^2$ indoors and $15 \text{mW}/\text{cm}^2$ outdoors. Single cells achieve a fairly stable output voltage of about 0.6 V (and have therefore to be used in series) as long as the drawn current does not exceed a critical threshold, which depends on the light intensity. Hence, solar cells are usually used to recharge secondary batteries.
- ☞ *Temperature gradients*: Differences in temperature can be directly converted to electrical energy.
- ☞ *Vibrations*: One almost pervasive form of mechanical energy is vibrations: walls or windows in buildings are resonating with cars or trucks passing in the streets, machinery often has low frequency vibrations. both amplitude and frequency of the vibration and ranges from about $0.1 \mu\text{W}/\text{cm}^3$ up to $10,000 \mu\text{W}/\text{cm}^3$ for some extreme cases. Converting vibrations to electrical energy can be undertaken by various means, based on electromagnetic, electrostatic, or piezoelectric principles.
- ☞ *Pressure variations*: Somewhat akin to vibrations, a variation of pressure can also be used as a power source.
- ☞ *Flow of air/liquid*: Another often-used power source is the flow of air or liquid in wind mills or turbines. The challenge here is again the miniaturization, but some of the work on millimeter scale MEMS gas turbines might be reusable.

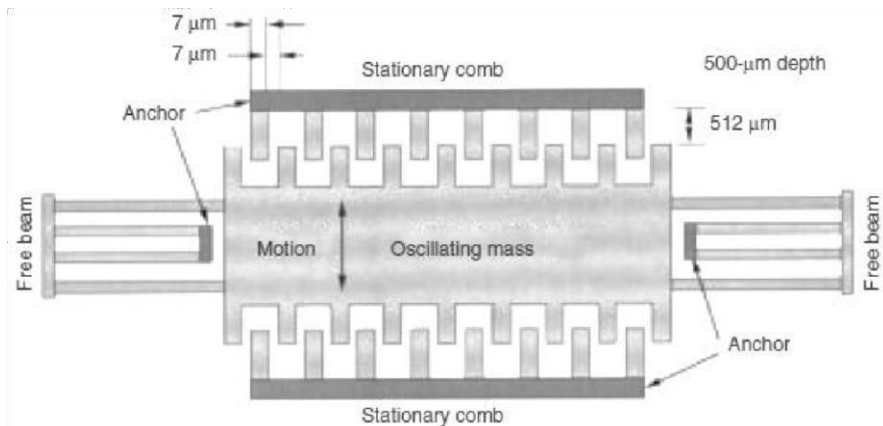


Figure 1.5 A MEMS device for converting vibrations to electrical energy, based on a variable capacitor

Energy source	Energy density
Batteries (zinc-air)	1050–1560 mWh/cm ³
Batteries (rechargeable lithium)	300 mWh/cm ³ (at 3–4 V)
Energy source	Power density
Solar (outdoors)	15 mW/cm ² (direct sun) 0.15 mW/cm ² (cloudy day)
Solar (indoors)	0.006 mW/cm ² (standard office desk) 0.57 mW/cm ² (<60 W desk lamp)
Vibrations	0.01–0.1 mW/cm ³
Acoustic noise	3 · 10 ⁻⁶ mW/cm ² at 75 dB 9, 6 · 10 ⁻⁴ mW/cm ² at 100 dB
Passive human-powered systems	1.8 mW (shoe inserts)
Nuclear reaction	80 mW/cm ³ , 10 ⁶ mWh/cm ³

TABLE 1.2: Comparison of energy sources

1.7 ENERGY CONSUMPTION OF SENSOR NODES:

In previous section we discussed about energy supply for a sensor node through batteries that have small capacity, and recharging by energy scavenging is complicated and volatile. Hence, the energy consumption of a sensor node must be tightly controlled. The main consumers of energy are the controller, the radio front ends, the memory, and type of the sensors. One method to reduce power consumption of these components is designing lowpower chips, it is the best starting point for an energy-efficient sensor node. But any advantages gained by such designs can easily be squandered/ wasted when the components are improperly operated. Second method for energy efficiency in wireless sensor node is reduced functionality by using multiple states of operation with reduced energy consumption. These modes can be introduced for all components of a sensor node, in particular, for controller, radio front end, memory, and sensors.

1.7.1 Microcontroller energy consumption: For a controller, typical states are “active”, “idle”, and “sleep”. A radio modem could turn transmitter, receiver, or both on or off. At time t_1 , the microcontroller is to be put into sleep mode should be taken to reduce power consumption from P_{active} to P_{sleep} . If it remains active and the next event occurs at time t_{event} , then a total energy is $E_{active} = P_{active} (t_{event} - t_1)$. On the other hand, requires a time τ_{down} until sleep mode has been reached. Let the average power consumption during this phase is $(P_{active} + P_{sleep})/2$. Then,

P_{sleep} is consumed until t_{event} . The energy saving is given by

$$E_{saved} = (t_{event} - t_1)P_{active} - (\tau_{down} (P_{active} + P_{sleep})/2 + (t_{event} - t_1 - \tau_{down})P_{sleep}) \quad (4)$$

Once the event to be processed occurs, however, an additional overhead of

$$E_{overhead} = \tau_{up} (P_{active} + P_{sleep})/2 \quad (5)$$

$$(t_{event} - t_1) > \frac{1}{2} \left(\tau_{down} + \frac{P_{active} + P_{sleep}}{P_{active} - P_{sleep}} \tau_{up} \right)$$

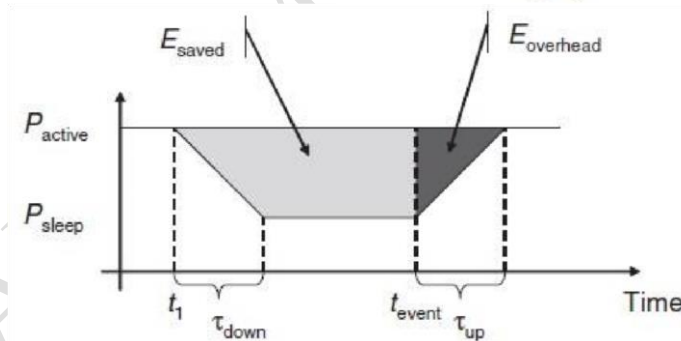


Figure 1.6 Energy savings and overheads for sleep modes

Switching to a sleep mode is only beneficial if $E_{overhead} < E_{saved}$ or, equivalently, if the time to the

next event is sufficiently large: ----- (6)

Examples:

Intel StrongARM

The Intel StrongARM provides three sleep modes:

- ✓ In *normal mode*, all parts of the processor are fully powered. Power consumption is up to 400 mW.
- ✓ In *idle mode*, clocks to the CPU are stopped; clocks that pertain to peripherals are active. Any interrupt will cause return to normal mode. Power consumption is up to 100 mW.
- ✓ In *sleep mode*, only the real-time clock remains active. Wakeup occurs after a timer interrupt and takes up to 160 ms. Power consumption is up to 50 μ W.

Texas Instruments MSP 430

The MSP430 family features a wider range of operation modes: One fully operational mode, which consumes about 1.2 mW (all power values given at 1 MHz and 3 V). There are four sleep modes in total. The deepest sleep mode, LPM4, only consumes 0.3 μ W, but the controller is only woken up by external interrupts in this mode. In the next higher mode, LPM3, a clock is also still running, which can be used for scheduled wake ups, and still consumes only about 6 μ W.

Atmel ATmega

The Atmel ATmega 128L has six different modes of power consumption, which are in principle similar to the MSP 430 but differ in some details. Its power consumption varies between 6 mW and 15 mW in idle and active modes and is about 75 μ W in power-down modes.

1.7.2 Memory energy consumption: The most relevant kinds of memory are on-chip memory and FLASH memory. Off-chip RAM is rarely used. In fact, the power needed to drive on-chip memory is usually included in the power consumption numbers given for the controllers. Hence, the most relevant part is FLASH memory. In fact, the construction and usage of FLASH memory can heavily influence node lifetime. The relevant metrics are the read and write times and energy consumption. Read times and read energy consumption tend to be quite similar between different types of FLASH memory. Energy consumption necessary for reading and writing to the Flash memory is used on the Mica nodes. Hence, writing to FLASH memory can be a time- and energy-consuming task that is best avoided if somehow possible.

1.7.3 Radio transceivers energy consumption: A radio transceiver has essentially two tasks: transmitting and receiving data between a pair of nodes. Similar to microcontrollers, radio transceivers can operate in different modes, the simplest ones are being turned on or turned off. To accommodate the necessary low total energy consumption, the transceivers should be turned off most of the time and only be activated when necessary – they work at a low duty cycle.

The energy consumed by a transmitter is due to two sources one part is due to RF signal generation, which mostly depends on chosen modulation and target distance. Second part is due to electronic components necessary for frequency synthesis, frequency conversion, filters, and so on. The transmitted power is generated by the amplifier of a transmitter. Its own power consumption P_{amp} depends on its architecture $P_{amp} = \alpha_{amp} + \beta_{amp}P_{tx}$. where α_{amp} and β_{amp} are constants depending on process technology and amplifier architecture. The energy to transmit a packet n -bits long (including all headers) then depends on how long it takes to send the packet, determined by the nominal bit rate R and the coding rate R_{code} , and on the total consumed power during transmission.

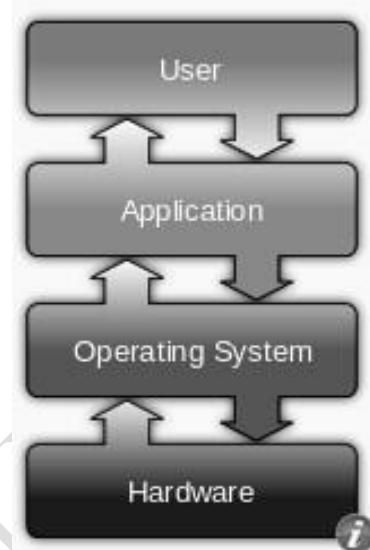
$$E_{tx}(n, R_{code}, P_{amp}) = T_{start}P_{start} + \frac{n}{RR_{code}}(P_{txElec} + P_{amp})$$

-- (7) Similar to the transmitter, the receiver can be either turned off or turned on. While being turned on, it can either actively receive a packet or can be idle, observing the channel and ready to receive. Evidently, the power consumption while it is turned off is negligible. Even the difference between idling and actually receiving is very small and can, for most purposes, be assumed to be zero. To elucidate, the energy E_{rcvd} required to receive a packet has a startup component $T_{start}P_{start}$ similar to the transmission case when the receiver had been turned off (startup times are considered equal for transmission and receiving here); it also has a component that is proportional to the packet time. During this time of actual reception, receiver circuitry has to be powered up, requiring a (more or less constant) power of P_{rxElec} .

(8)

1.7.4 Power consumption of sensor and actuators:

Providing any guidelines about the power consumption of the actual sensors and actuators is impossible because of the wide variety of these devices. For example, passive light or temperature sensors – the power consumption can possibly be ignored in comparison to other devices on a wireless node. For others, active devices like sonar(A measuring instrument that sends out an acoustic pulse in water and measures distances in terms of time for the echo of the pulse to return), power consumption can be quite considerable in the dimensioning of power sources on the sensor node, not to overstress batteries.



Unit 2

1.8 OPERATING SYSTEMS AND EXECUTION ENVIRONMENTS:

1.8.1 Embedded operating systems:

- ✓ An operating system (OS) is system software that manages computer hardware and software resources and provides common services for computer programs.
- ✓ For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware.
- ✓ An embedded system is some combination of computer hardware and software, either fixed in capability or programmable, that is specifically designed for a particular function.
- ✓ Embedded operating systems are designed to be used in embedded computer systems. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design.

Figure 1.7

Operating

Systems

1.8.2 TinyOS:

- ✓ TinyOS is an open-source, flexible and application-specific operating system for wireless sensor networks.
- ✓ Wireless sensor network consists of a large number of tiny and low-power nodes, each of which executes simultaneous and reactive programs that must work with strict memory and power constraints.
- ✓ TinyOS meets these challenges and has become the platform of choice for sensor network such as limited resources and low-power operation.
- ✓ Salient features of TinyOS are

- ☞ A simple event-based concurrency model and split-phase operations that influence the development phases and techniques when writing application code.
- ☞ It has a component-based architecture which provides rapid innovation and implementation while reducing code size as required by the difficult memory constraints inherent in wireless sensor networks.
- ☞ TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools.
- ☞ TinyOS's event-driven execution model enables fine grained power management, yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces.

1.8.3 Programming paradigms and application programming interfaces:

- ❖ **Concurrent Programming:** Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Concurrent means something that happens at the same time as something else. Tasks are broken down into subtasks that are then assigned to separate processors to simultaneously, instead of sequentially as they would have to be carried out by a single processor. Concurrent processing is sometimes said to be synonymous with parallel processing.
- ❖ **Process-based concurrency:** Most general-purpose operating systems concurrent (seemingly parallel) execution multiple processes on a single CPU. Using processes you are forced to deal with communication through messages, which is the Erlang(A unit of traffic intensity in telephone system) way of doing communication. Data is not shared, so there is no risk of data corruption. Fault-tolerance scalability is the main advantages of processes vs. threads. Another

advantage processes is that they can crash and you are perfectly ok with that, because you just restart them (even across network hosts). If thread crashes, it may crash the entire process, which may bring down your entire application.

- ❖ **Event-based programming:** In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key

presses), sensor outputs, or messages from other programs/threads. Event-driven programming is the dominant paradigm used in Graphical User Interfaces (GUI type of user interface that allows users to interact with electronic devices through graphical icons) and other applications. The system essentially waits for any event to happen, where an event typically can be the availability of data from a sensor, the arrival of a packet, or the expiration of a timer. Such an event is then handled by a short sequence of instructions that only stores the fact that this event has occurred and stores the necessary information.

- ❖ **Interfaces to the operating system:** A boundary across which two independent systems meet and act on or communicate with each other. In computer technology, there are several types of interfaces. User interface - the keyboard, mouse, menus of a computer system. The user interface allows the user to communicate with the operating system. Stands for "Application Programming Interface." An API is a set of commands, functions, protocols, and objects (wireless links, nodes) that programmers can use to create software or interact with an external system (sensors, actuators, transceivers). It provides developers with standard commands for performing common operations so they do not have to write the code from scratch.

1.8.4 Structure of operating system and protocol stack:

The traditional approach to communication protocol structuring is to use layering: individual protocols are

perform

modern,
support

of **Figure 1.8** Sequential program

Handle sensor process



and
using
of

stacked on top of each other, each layer only using functions of the layer directly. This layered approach has great benefits in keeping the entire protocol stack manageable, in containing complexity, and in promoting modularity and reuse. For the purposes of a WSN, however, it is not clear whether such a strictly layered approach will serve. A protocol stack refers to a group of protocols that are running concurrently that are employed for the implementation of network protocol suite. The protocols in a stack determine the interconnectivity rules for a layered network model such as in the OSI or TCP/IP models.

1.8.5 Dynamic energy and power management: Switching individual components into various sleep states or reducing their performance by scaling down frequency and supply voltage and selecting particular modulation and coding are prominent examples for improving energy efficiency. To control these possibilities, decisions have to be made by the operating system, by the protocol stack, or potentially by an application when to switch into one of these states. Dynamic Power Management (DPM) on a system level is the problem at hand. One of the complicating factors to DPM is the energy and time required for the transition of a component between any two states. If these factors were negligible, clearly it would be optimal to always & immediately go into the mode with the lowest power consumption possible.

NETWORK ARCHITECTURE: It introduces the basic principles of turning individual sensor nodes into a wireless sensor network. In this optimization goals of how a network should function are discussed as

- ✓ Sensor network scenarios
- ✓ Optimization goals and figures of merit
- ✓ Gateway concepts

1.9 SENSOR NETWORK SCENARIOS:

1.9.1 Types of sources and sinks: Source is any unit in the network that can provide information (sensor node). A sink is the unit where information is required, it could belong to the sensor network or outside this network to interact with another network or a gateway to another larger Internet. Sinks are illustrated by Figure 1.11, showing sources and sinks in direct communication.

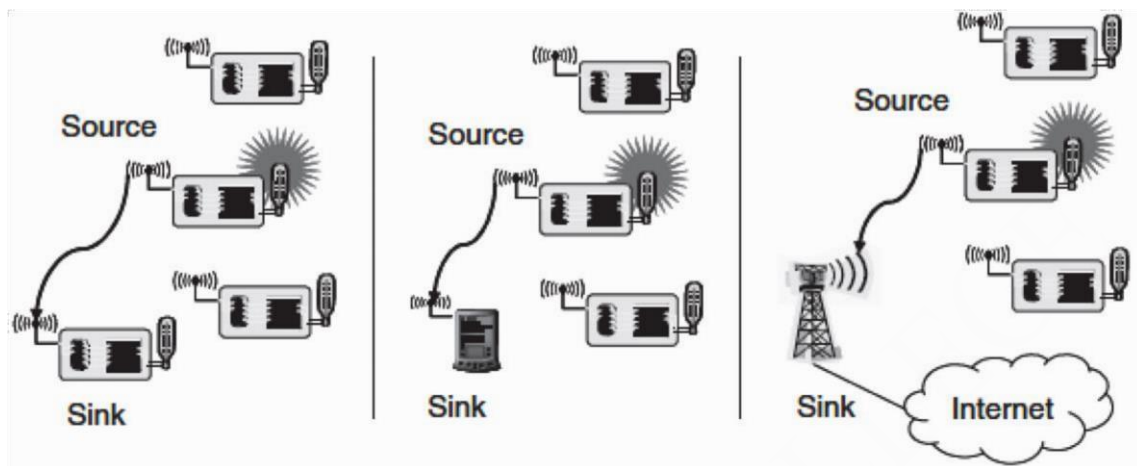


Figure 1.11 Three types of sinks in a very simple, single-hop sensor network

1.9.2 Single-hop versus multi-hop networks:

Because of limited distance the direct communication between source and sink is not always possible. In WSNs, to cover a lot of environment the data packets taking multi hops from source to the sink. To overcome such limited distances it better to use relay stations, The data packets taking multi hops from source to the sink as shown in Figure 1.12, Depending on the particular application of having an intermediate sensor node at the right place is high.

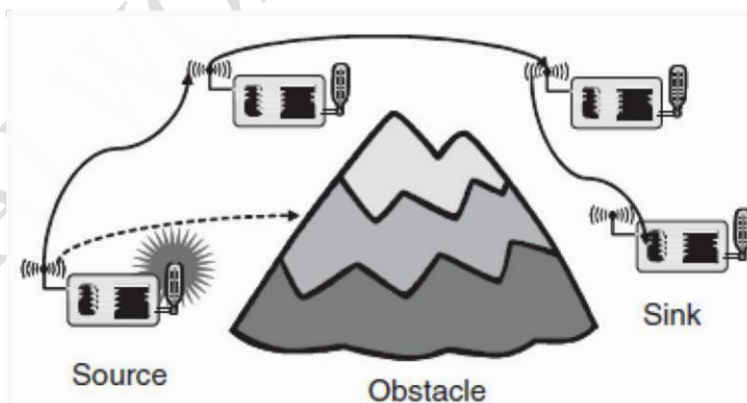


Figure 1.12 Multi-hop networks: As direct communication is impossible because of distance and/or obstacles

Multi-hopping also improves the energy efficiency of communication as it consumes less energy to use relays instead of direct communication, the radiated energy required for

direct communication over a distance d is cd^α (c some constant, $\alpha \geq 2$ the path loss coefficient) and using a relay at distance $d/2$ reduces this energy to $2c(d/2)^\alpha$

This calculation considers only the radiated energy. It should be pointed out that only multihop networks operating in a store and forward fashion are considered here. In such a network, a node has to correctly receive a packet before it can forward it somewhere. Cooperative relaying (reconstruction in case of erroneous packet reception) techniques are not considered here.

1.9.3 Multiple sinks and sources: In many cases, multiple sources and multiple sinks present. Multiple sources should send information to multiple sinks. Either all or some of the information has to reach all or some of the sinks. This is illustrated in figure 1.13.

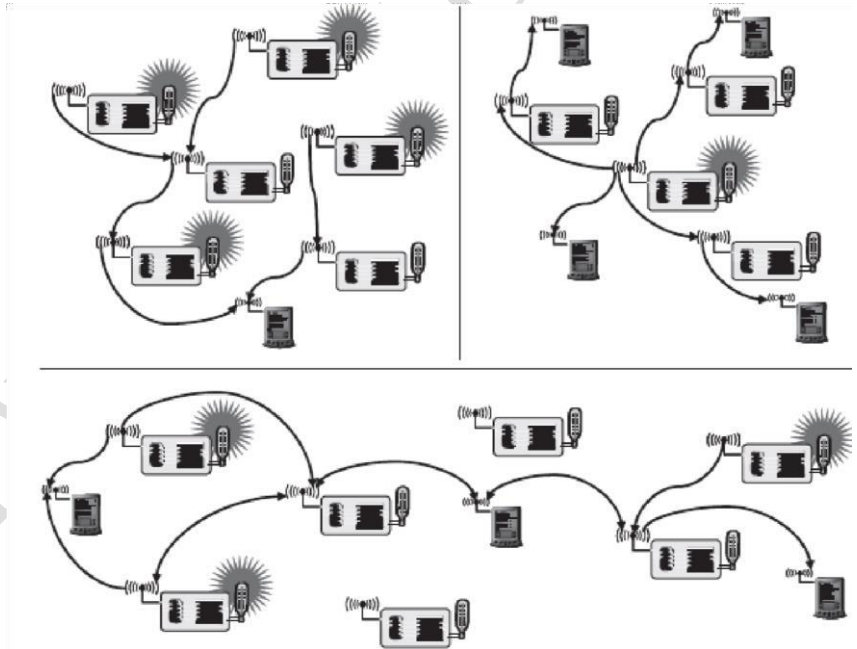


Figure 1.13 Multiple sources and/or multiple sinks.

Note how in the scenario in the lower half, both sinks and active sources are used to forward data to the sinks at the left and right end of the network.

1.9.4 Three types of mobility: In the scenarios discussed above, all participants were stationary. But one of the main virtues of wireless communication is its ability to support mobile

participants In wireless sensor networks, mobility can appear in three main forms a. Node mobility

b. Sink mobility

c. Event mobility

1.9.4(a) Node Mobility: The wireless sensor nodes themselves can be mobile. The meaning of such mobility is highly application dependent. In examples like environmental control, node mobility should not happen; in livestock surveillance (sensor nodes attached to cattle, for example), it is the common rule. In the face of node mobility, the network has to reorganize to function correctly.

1.9.4(b) Sink Mobility: The information sinks can be mobile. For example, a human user requested information via a PDA while walking in an intelligent building. In a simple case, such a requester can interact with the WSN at one point and complete its interactions before moving on. In many cases, consecutive interactions can be treated as separate, unrelated requests.

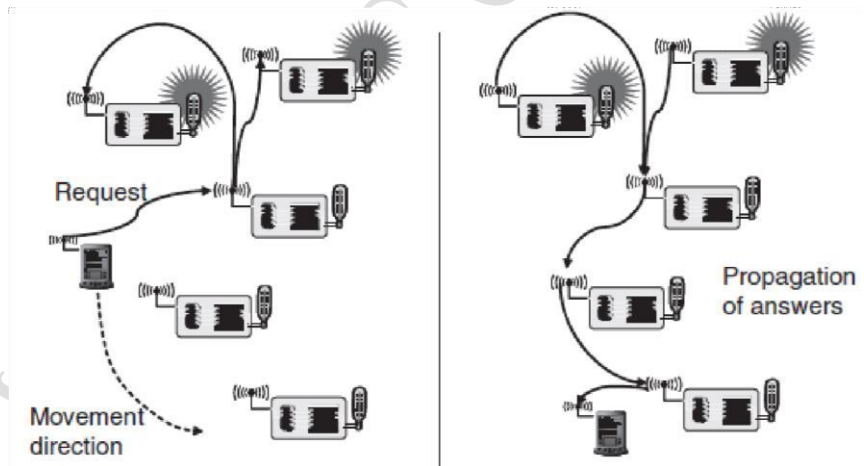


Figure 1.14 Sink mobility: A mobile sink moves through a sensor network as information is being retrieved *on its behalf*

1.9.4(c) Event Mobility: In tracking applications, the cause of the events or the objects to be tracked can be mobile. In such scenarios, it is (usually) important that the observed event is covered by a sufficient number of sensors at all time. As the event source moves through the network, it is accompanied by an area of activity within the network – this has been called the frisbee model. This notion is described by Figure 1.15, where the task is to detect a moving elephant and to observe it as it moves around

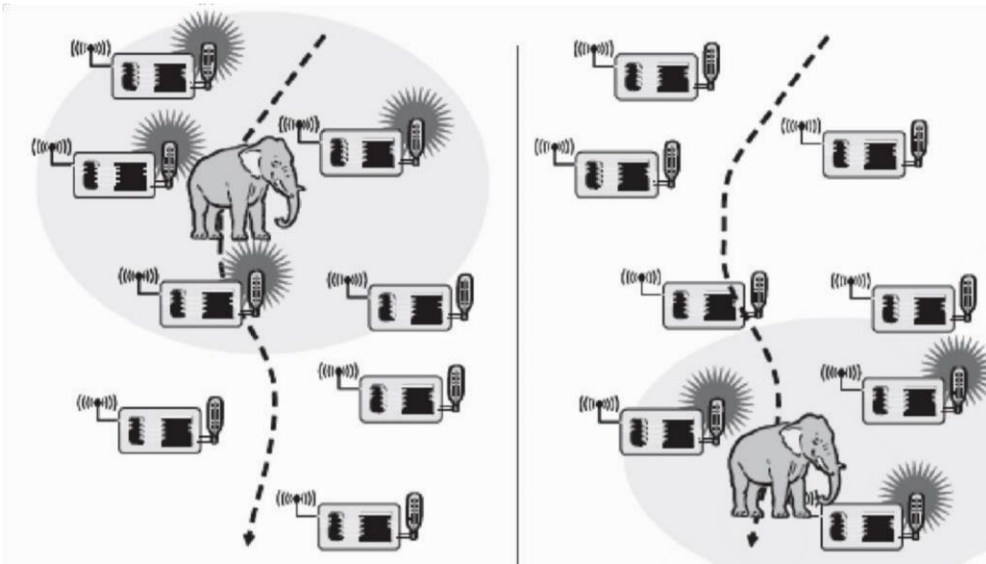


Figure 1.15 Area of sensor nodes detecting an event – an elephant– that moves through the network along with the event source (dashed line indicate the elephant’s trajectory; shaded ellipse the activity area following or even preceding the elephant)

1.10 OPTIMIZATION GOALS AND FIGURES OF MERIT:

For all WSN scenarios and application types have to face the challenges such as How to optimize a network and How to compare these solutions?

- ✓ How to decide which approach is better?
- ✓ How to turn relatively inaccurate optimization goals into measurable figures of merit?

For all the above questions the general answer is obtained from

- ❖ Quality of service
- ❖ Energy efficiency
- ❖ Scalability
- ❖ Robustness

1.10.1 Quality of service: WSNs differ from other conventional communication networks in the type of service they offer. These networks essentially only move bits from one place to another. Some generic possibilities are

- ✓ **Event detection/reporting probability-** The probability that an event that actually occurred is not detected or not reported to an information sink that is interested in such an event For example, not reporting a fire alarm to a surveillance station would be a severe shortcoming.

- ✓ **Event classification error-** If events are not only to be detected but also to be classified, the error in classification must be small
- ✓ **Event detection delay** -It is the delay between detecting an event and reporting it to any/all interested sinks
- ✓ **Missing reports** -In applications that require periodic reporting, the probability of undelivered reports should be small
- ✓ **Approximation accuracy-** For function approximation applications, the average/maximum absolute or relative error with respect to the actual function.
- ✓ **Tracking accuracy** Tracking applications must not miss an object to be tracked, the reported position should be as close to the real position as possible, and the error should be small.

1.10.2 Energy efficiency: Energy efficiency should be optimization goal. The most commonly considered aspects are:

- ✓ **Energy per correctly received bit-**How much energy is spent on average to transport one bit of information (payload) from the transmitter to the receiver.
- ✓ **Energy per reported (unique) event-**What is the average energy spent to report one event
- ✓ **Delay/energy trade-offs-**“urgent” events increases energy investment for a speedy reporting events. Here, the trade-off between delay and energy overhead is interesting
- ✓ **Network lifetime** The time for which the network is operational
- ✓ **Time to first node death-**When does the first node in the network run out of energy or fail and stop operating?
- ✓ **Network half-life-**When have 50 % of the nodes run out of energy and stopped operating
- ✓ **Time to partition-**When does the first partition of the network in two (or more) disconnected parts occur?
- ✓ **Time to loss of coverage** the time when for the first time any spot in the deployment region is no longer covered by any node’s observations.
- ✓ **Time to failure of first event notification** A network partition can be seen as irrelevant if the unreachable part of the network does not want to report any events in the first place.

1.10.3 Scalability: The ability to maintain performance characteristics irrespective of the size of the network is referred to as scalability. With WSN potentially consisting of thousands of nodes, scalability is an obviously essential requirement. The need for extreme scalability has direct consequences for the protocol design. Often, a penalty in performance or complexity has to be paid for small networks. Architectures and protocols should implement appropriate scalability support rather than trying to be as scalable as possible. Applications with a few dozen nodes might admit more-efficient solutions than applications with thousands of nodes.

1.10.4 Robustness: Wireless sensor networks should also exhibit an appropriate robustness. They should not fail just because a limited number of nodes run out of

energy, or because their environment changes and severs existing radio links between two nodes. If possible, these failures have to be compensated by finding other routes.

1.11 GATE WAY CONCEPTS:

1.11.1 Need for gateways:

- ✓ For practical deployment, a sensor network only concerned with itself is insufficient.
- ✓ The network rather has to be able to interact with other information devices for example to read the temperature sensors in one's home while traveling and accessing the Internet via a wireless.
- ✓ Wireless sensor networks should also exhibit an appropriate robustness
- ✓ They should not fail just because of a limited number of nodes run out of energy or because of their environment changes and breaks existing radio links between two nodes.
- ✓ If possible, these failures have to be compensated by finding other routes. Figure 1.16 shows this networking scenario, The WSN first of all has to be able to exchange data with such a mobile device or with some sort of gateway, which provides the physical connection to the Internet. The WSN support standard wireless communication technologies such as IEEE 802.11. The design of gateways becomes much more challenging when considering their logical design. One option is to regard a gateway as a simple router between Internet and sensor network.

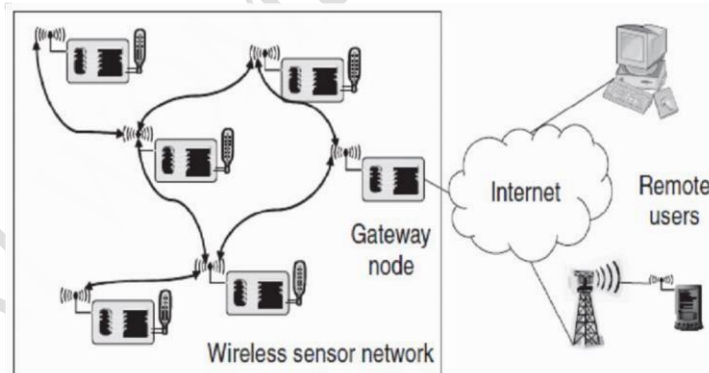


Figure 1.16 A wireless sensor network with gateway node, enabling access to remote clients via the Internet

1.11.2 WSN to Internet communication: Assume that the initiator of a WSN – Internet communication resides in the WSN.

- ✓ For example, a sensor node wants to deliver an alarm message to some Internet host.
- ✓ The first problem to solve is how to find the gateway from within the network
- ✓ Basically, a routing problem to a node that offers a specific service has to be solved, integrating routing and service discovery

- ✓ If several such gateways are available, how to choose between them?
- ✓ In particular, if not all Internet hosts are reachable via each gateway or at least if some gateway should be preferred for a given destination host?
- ✓ How to handle several gateways, each capable of IP networking, and the communication among them?
- ✓ One option is to build an IP overlay network on top of the sensor network – How to map a semantic notion (“Alert Alice”) to a concrete IP address?
- ✓ Even if the sensor node does not need to be able to process the IP protocol, it has to include sufficient information (IP address and port number, for example) in its own packets;
- ✓ the gateway then has to extract this information and translate it into IP packets.
- ✓ An ensuing question is which source address to use here – the gateway in a sense has to perform tasks similar to that of a Network Address Translation (NAT) device.

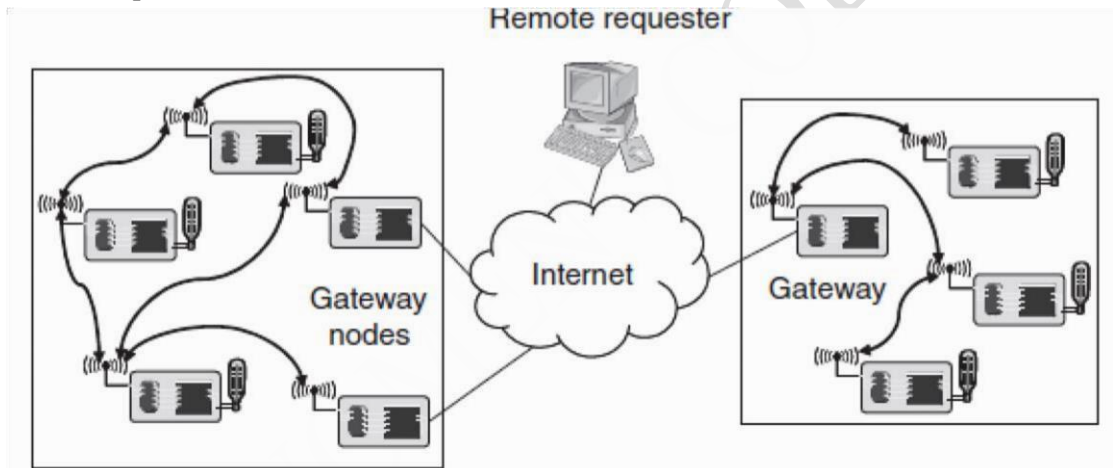
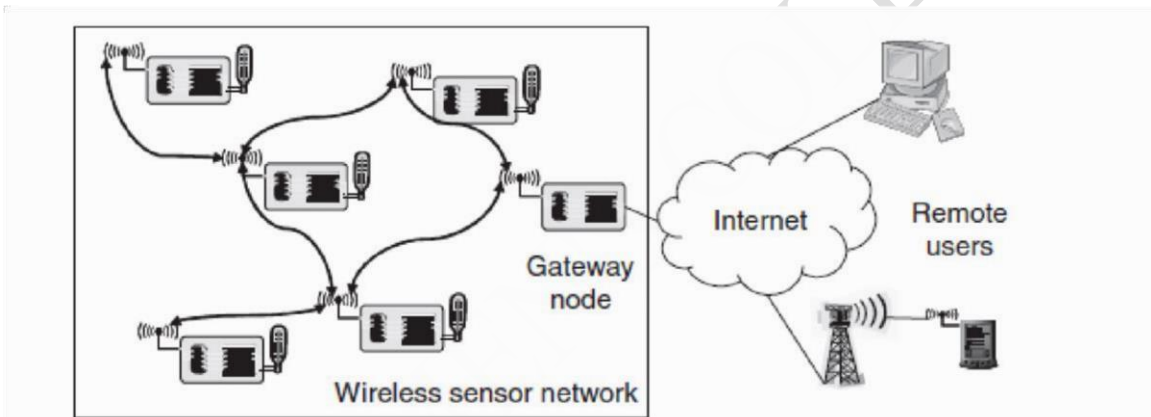


Figure 1.17: A wireless Sensor Network with gateway node, enabling access to remote clients via the WSN

1.11.3 Internet to WSN communication: The case of an Internet-based entity trying to access services of a WSN is even more challenging.

- ✓ This is fairly simple if this requesting terminal is able to directly communicate with the WSN.
- ✓ The more general case is, however, a terminal “far away” requesting the service, not immediately able to communicate with any sensor node and thus requiring the assistance of a gateway node
- ✓ First of all, again the question is how to find out that there actually is a sensor network in the desired location, and how to find out about the existence of a gateway node?
- ✓ Once the requesting terminal has obtained this information, how to access the actual services.

- ✓ The requesting terminal can instead send a properly formatted request to this gateway, which acts as an application-level gateway
- ✓ The gateway translates this request into the proper intra sensor network protocol interactions
- ✓ The gateway can then mask, for example, a data-centric data exchange within the network behind an identity-centric exchange used in the Internet
- ✓ It is by no means clear that such an application-level protocol exists that represents an actual simplification over just extending the actual sensor network protocols to the remote terminal
- ✓ In addition, there are some clear parallels for such an application-level protocol with so-called Web Service Protocols, which can explicitly describe services and the way



A wireless sensor network with gateway node, enabling access to remote clients via the Internet. The gateway node acts as a bridge between the WSN and the Internet, allowing remote users to access the network.

Figure 1.18: A wireless Sensor Network with gateway node, enabling access to remote clients via the internet

1.11.4 WSN tunnelling:

- ✓ The gateways can also act as simple extensions of one WSN to another WSN. The idea is to build a larger, “virtual” WSN out of separate parts, transparently “tunneling” all protocol messages between these two networks and simply using the Internet as a transport network.
- ✓ This can be attractive, but care has to be taken not to confuse the virtual link between two gateway nodes with a real link;
- ✓ Otherwise, protocols that rely on physical properties of a communication link can get quite confused (e.g. time synchronization or localization protocols).

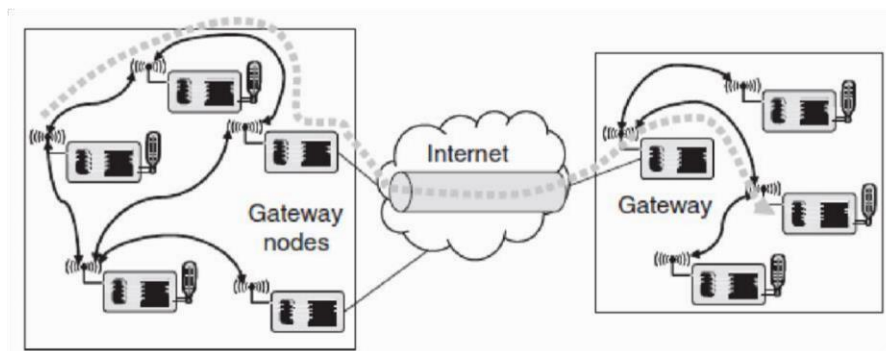


Figure 1.19 Connecting two WSNs with a tunnel over the Internet

UNIT-III

MAC PROTOCOLS FOR WIRELESS SENSOR NETWORKS

3.1 INTRODUCTION:

Nodes in an Ad-hoc wireless network share a common broadcast radio channel. Since the radio spectrum is limited, the bandwidth available for communication in such networks is also limited. Access to this shared medium should be controlled in such a manner that all nodes receive a fair share of the available bandwidth, and that the bandwidth is utilized efficiently. Characteristics of the wireless medium are completely different from wired medium. So a different set of protocols is required for controlling access to the shared medium in such networks. This is achieved by using Medium Access Control (MAC) protocol.

3.2 ISSUES IN DESIGNING A MAC PROTOCOL FOR AD HOC WIRELESS NETWORKS

The following are the main issues that need to be addressed while designing a MAC protocol for Ad-hoc wireless networks.

3.2.1 Bandwidth Efficiency: Since the radio spectrum is limited, the bandwidth available for communication is also very limited. The MAC protocol must be designed in such a way that to maximize this bandwidth efficiency (the ratio of the bandwidth

used for actual data transmission to the total available bandwidth). That is the uncommon bandwidth is utilized in an efficient manner.

3.2.3 Quality of Service Support(QoS): Providing QoS support to data sessions in Adhoc networks is very difficult due to their characteristic nature of nodes mobility. Most of the time, Bandwidth reservation made at one point of time may become invalid once the node moves out of the region. The MAC protocol for Ad-hoc wireless networks that are to be used in such real-time applications must have resource reservation mechanism take care of nature of the wireless channel and the mobility of nodes.

3.2.4 Synchronization: The MAC protocol must take into consideration the synchronization between nodes in the network. Synchronization is very important for bandwidth (time slot) reservations by nodes achieved by exchange of control packets.

3.2.5 Hidden and Exposed Terminal Problems: The hidden terminal problem refers to the collision of packets at a receiving node due to the simultaneous transmission of those nodes. The exposed terminal problem refers to the inability of a node, which is blocked due to transmission by a nearby transmitting node, to transmit to another node.

3.2.6 Mobility of Nodes: This is a very important factor affecting the performance (throughput) of the protocol. The MAC protocol obviously has no role to play in influencing the mobility of the nodes.

3.2.7 Error-Prone Shared Broadcast Channel: Due to broadcast nature of the radio channel (transmissions made by a node are received by all nodes within its direct transmission range) there is a possibility of packet collisions is quite high in wireless networks. A MAC protocol should grant channel access to nodes in such a manner that collisions are minimized.

3.2.8 Distributed Nature/Lack of Central Coordination: Ad hoc wireless networks do not have centralized coordinators because nodes keep moving continuously. Therefore, nodes must be scheduled in a distributed fashion for gaining access to the channel. The MAC protocol must make sure that the additional overhead, in terms of bandwidth consumption is not very high.

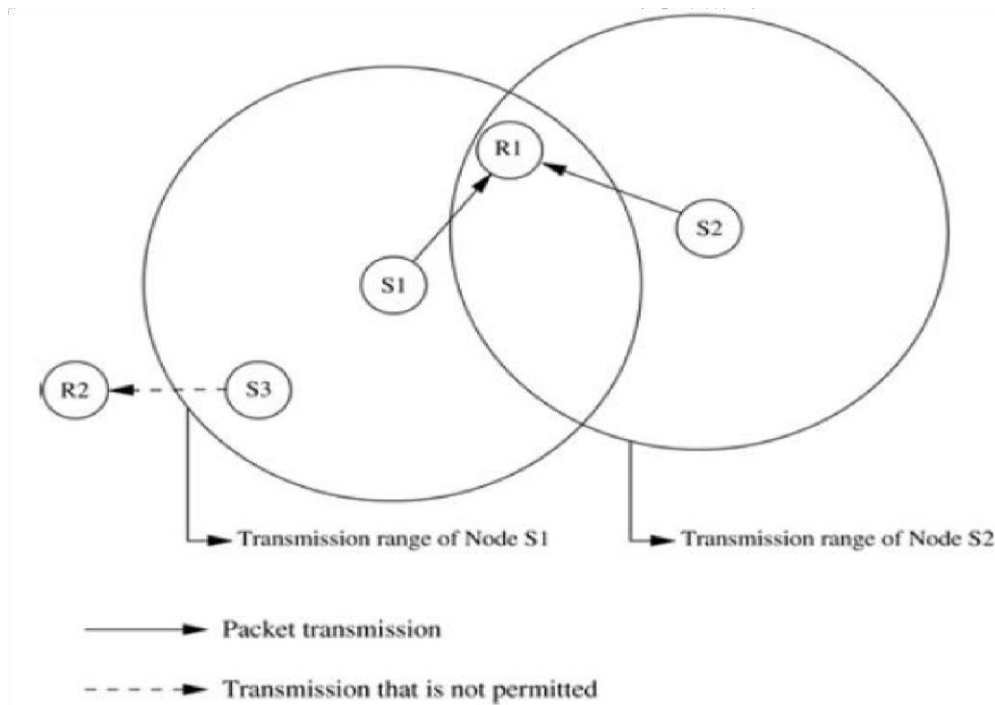


Figure 3.1 Hidden and Exposed node problems

3.3 DESIGN GOALS OF A MAC PROTOCOL FOR AD-HOC WIRELESS NETWORKS:

The following are the important goals to be met while designing a medium access control (MAC) protocol for ad hoc wireless networks:

1. The operation of the protocol should be distributed.
2. The protocol should provide QoS support for real-time traffic.
3. The access delay, which refers to the average delay experienced by any packet to get transmitted, must be kept low.
4. The available bandwidth must be utilized efficiently.
5. The protocol should ensure fair allocation of bandwidth to nodes.
6. Control overhead must be kept as low as possible.
7. The protocol should minimize the effects of hidden and exposed terminal problems.
8. The protocol must be scalable to large networks.
9. It should have power control mechanisms.
10. The protocol should have mechanisms for adaptive data rate control.
11. It should try to use directional antennas.
12. The protocol should provide synchronization among nodes.

3.4 CLASSIFICATIONS OF MAC PROTOCOLS:

MAC protocols for ad hoc wireless networks can be classified into several categories based on various criteria such as initiation approach, time synchronization, and reservation approaches. Ad hoc network MAC protocols can be classified into three basic types:

- a. Contention-based protocols
- b. Contention-based protocols with reservation mechanisms
- c. Contention-based protocols with scheduling mechanisms

Apart from these three major types, there exist other MAC protocols that cannot be classified clearly under any one of the above three types of protocols.

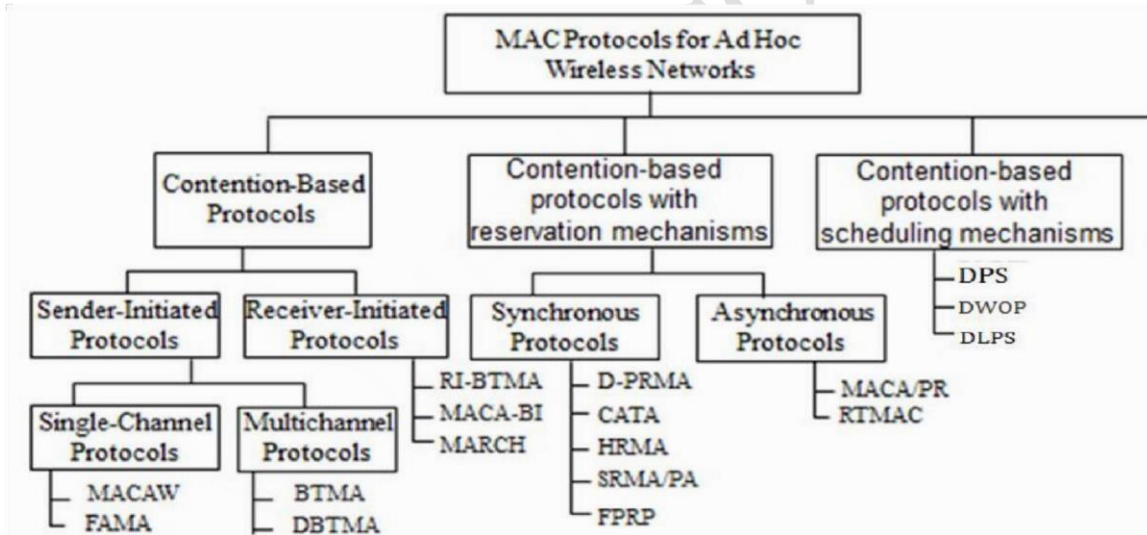


Figure 3.2 Classifications of MAC Protocols

3.4.1 Contention-based protocols:

These protocols follow a contention-based channel access policy. Whenever it receives a packet to be transmitted, it contends with its neighbor nodes for access to the shared channel. Contention-based protocols cannot provide QoS guarantees to sessions since nodes are not guaranteed regular access to the channel. Random access protocols can be further divided into two types:

1. *Sender-initiated protocols:* Packet transmissions are initiated by the sender node.

2. *Receiver-initiated protocols*: The receiver node initiates the contention resolution protocol. Sender-initiated protocols can be further divided into two types:
 - a. *Single-channel sender-initiated protocols*: In these protocols, the total available bandwidth is used as it is, without being divided. A node that wins the contention to the channel can make use of the entire bandwidth.
 - b. *Multichannel sender-initiated protocols*: In multichannel protocols, the available bandwidth is divided into multiple channels. This enables several nodes to simultaneously transmit data, each using a separate channel. Some protocols dedicate a frequency channel exclusively for transmitting control information.

3.4.2 Contention-Based Protocols with Reservation Mechanisms Ad hoc wireless networks sometimes may need to support real-time traffic, which requires QoS guarantees to be provided. In contention-based protocols, nodes are not guaranteed periodic access to the channel. Hence they cannot support real-time traffic. In order to support such traffic, certain protocols have mechanisms for reserving bandwidth a priori. Such protocols can provide QoS support to time-sensitive traffic sessions. These protocols can be further classified into two types:

1. *Synchronous protocols*: Synchronous protocols require time synchronization among all nodes in the network, so that reservations made by a node are known to other nodes in its neighborhood. Global time synchronization is generally difficult to achieve.
2. *Asynchronous protocols*: They do not require any global synchronization among nodes in the network. These protocols usually use relative time information for effecting reservations

3.4.3 Contention-Based Protocols with Scheduling Mechanisms

Node scheduling is done in a manner so that all nodes are treated fairly and no node is starved of bandwidth. Scheduling-based schemes are also used for enforcing priorities among flows whose packets are queued at nodes. Some scheduling schemes also take into consideration battery characteristics, such as remaining battery power, while scheduling nodes for access to the channel.

3.4.4 Other Protocols

There are several other MAC protocols that do not strictly fall under the above categories.

3.5 CONTENTION-BASED PROTOCOLS:

These protocols follow a contention-based channel access policy. Whenever it receives a packet to be transmitted, it contends with its neighbor nodes for access to the shared channel. Contention-based protocols cannot provide QoS guarantees to sessions since

nodes are not guaranteed regular access to the channel. Random access protocols can be further divided into two types:

1. *Sender-initiated protocols*: Packet transmissions are initiated by the sender node.
2. *Receiver-initiated protocols*: The receiver node initiates the contention resolution protocol.

3.5.1 Sender-initiated protocols : These are further divided into two types:

I. Single-channel sender-initiated protocols: In these protocols, the total available bandwidth is used as it is, without being divided. A node that wins the contention to the channel can make use of the entire bandwidth. *Examples*: MACAW, FAMA

a) MACAW: Multiple Access Collision

Avoidance for Wireless A Media Access Protocol for Wireless LANs is based on MACA Protocol.

MACA Protocol:

- When a node wants to transmit a data packet, it first transmit a RTS (Request To Send) frame.
 - The receiver node, on receiving the RTS packet, if it is ready to receive the data packet, transmits a CTS (Clear to Send) packet.
 - Once the sender receives the CTS packet without any error, it starts transmitting the data packet.
 - If a packet transmitted by a node is lost, the node uses the Binary Exponential Back-off (BEB) algorithm to back-off a random interval of time before retrying.
- The problem is solved by MACAW.

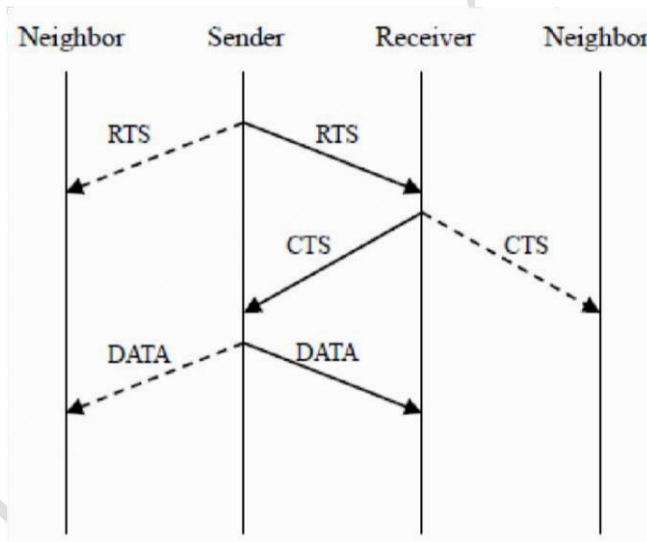
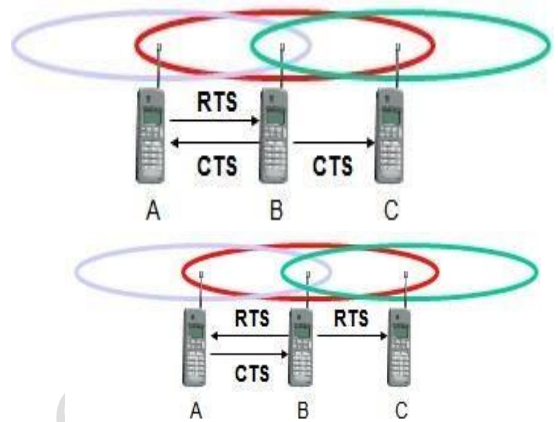


Figure 3.3 Packet transmission in MACA

MACA EXAMPLES:

1. MACA avoids the problem of hidden terminals

- ✓ A and C want to send to B
- ✓ A sends RTS first
- ✓ C waits after receiving CTS from B

2. MACA avoids the problem of exposed terminals

- ✓ B wants to send to A, C to another

- terminal
- ✓ Now C does not have to wait for it cannot receive CTS from A

MACAW Protocol:

MACA for Wireless is a revision of MACA.

- ✓ *The sender* transmits a **RTS (Request To Send)** frame if no nearby station transmits a RTS.
- ✓ *The receiver* replies with a **CTS (Clear To Send)** frame.
- ✓ *Neighbors*
 - Can see CTS, then keep quiet.
 - Can see RTS but not CTS, then keep quiet until the CTS is back to the sender.
- ✓ The receiver sends an **ACK** when receiving a frame.
 - Neighbors keep silent until see ACK.
- ✓ *Collisions*
 - There is no collision detection.
 - The senders know collision when they don't receive CTS.
 - They each wait for the exponential backoff time.

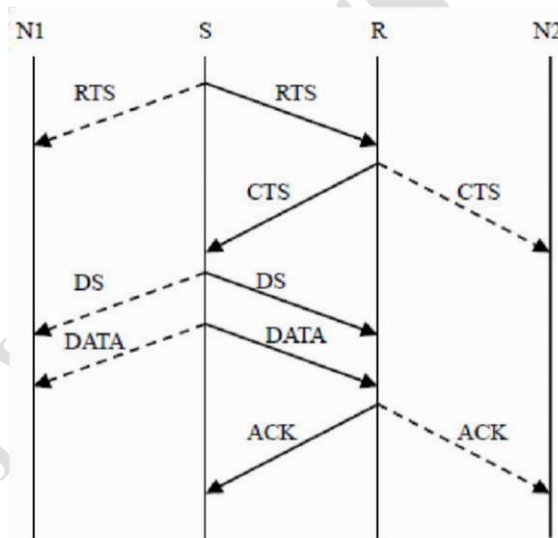


Figure 3.4 Packet transmission in MACAW

b) FAMA: Floor Acquisition Multiple Access Protocols

The floor acquisition multiple access (FAMA) protocols are based on a channel access discipline which consists of a carrier-sensing operation and a collision-avoidance dialog between the sender and the intended receiver of a packet. Floor acquisition refers to the process of gaining control of the channel. At any given point of time, the control of the channel is assigned to only one node, and this node is guaranteed to transmit one or more data packets to different destinations without suffering from packet collisions. Carrier-

sensing by the sender, followed by the RTS-CTS control packet exchange, enables the protocol to perform as efficiently as MACA in the presence of hidden terminals, and as efficiently as CSMA otherwise.

FAMA requires a node that wishes to transmit packets to first acquire the floor (channel) before starting to transmit the packets. The floor is acquired by means of exchanging control packets. Though the control packets themselves may collide with other control packets, it is ensured that data packets sent by the node that has acquired the channel are always transmitted without any collisions. Any single-channel MAC protocol that does not require a transmitting node to sense the channel can be adapted for performing floor acquisition tasks. Floor acquisition using the RTS-CTS exchange is advantageous as the mechanism also tries to provide a solution for the hidden terminal problem. There are two FAMA protocol variants available:

- RTS-CTS exchange with no carrier sensing (MACA).
- RTS-CTS exchange with non-persistent carrier-sensing (FAMA-NTR).

RTS-CTS exchange with no carrier sensing (MACA): In MACA, a ready node transmits an RTS packet. A neighbor node receiving the RTS defers its transmissions for the period specified in the RTS. On receiving the RTS, the receiver node responds by sending back a CTS packet, and waits for a long enough period of time in order to receive a data packet. Neighbor nodes of the receiver which hear this CTS packet defer their transmissions for the time duration of the impending data transfer. In MACA, nodes do not sense the channel. A node defers its transmissions only if it receives an RTS or CTS packet. In MACA, data packets are prone to collisions with RTS packets.

FAMA – Non-Persistent Transmit Request: Before sending a packet, the sender senses the channel. If channel is busy, the sender back-off a random time and retries later. If the channel is free, the sender sends RTS and waits for a CTS packet. If the sender cannot receive a CTS, it takes a random back-off and retries later. If the sender receives a CTS, it can start transmission data packet. In order to allow the sender to send a burst of packets, the receiver is made to wait a time duration τ seconds after a packet is received.

Multichannel sender-initiated protocols: In multichannel protocols, the available bandwidth is divided into multiple channels. This enables several nodes to simultaneously transmit data, each using a separate channel. Some protocols dedicate a frequency channel exclusively for transmitting control information.

II. *Multi-channel sender-initiated protocols:*

a) Busy Tone Multiple Access Protocols (BTMA):

The Busy Tone Multiple Access (BTMA) protocol is one of the earliest protocols proposed for overcoming the hidden terminal problem faced in wireless environments. The transmission channel is split into two: a data channel and a control channel. The data channel is used for data packet transmissions, while the control channel is used to transmit the busy tone signal.

When a node is ready for transmission, it senses the channel to check whether the busy tone is active. If not, it turns on the busy tone signal and starts data transmission; otherwise, it reschedules the packet for transmission after some random rescheduling delay. Any other node which senses the carrier on the incoming data channel also transmits the busy tone signal on the control channel. Thus, when a node is transmitting, no other node in the two-hop neighborhood of the transmitting node is permitted to simultaneously transmit. Though the probability of collisions is very low in BTMA, the bandwidth utilization is very poor. Figure 3.5 shows the worstcase scenario where the node density is very high; the dotted circle shows the region in which nodes are blocked from simultaneously transmitting when node N1 is transmitting packets.

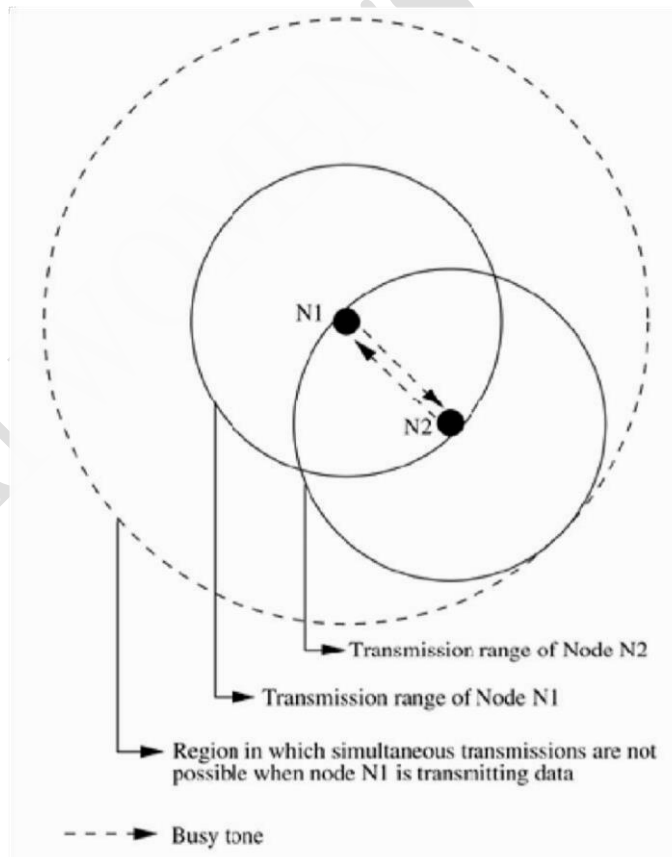


Figure 3.5 Packet transmission in BTMA

b) Dual Busy Tone Multiple Access Protocol (DBTMAP):

It is an extension of the BTMA scheme. In this also transmission channel is split in to two parts. A data channel for data packet transmissions and a control channel used for control packet transmissions (RTS and CTS packets) and also for transmitting the busy tones. In this protocol use two busy tones on the control channel, BT_t and BT_r . Where BT_t indicate that it is transmitting on the data channel and BT_r indicate that it is receiving on the data channel. Two busy tone signals are two sine waves at different frequencies.

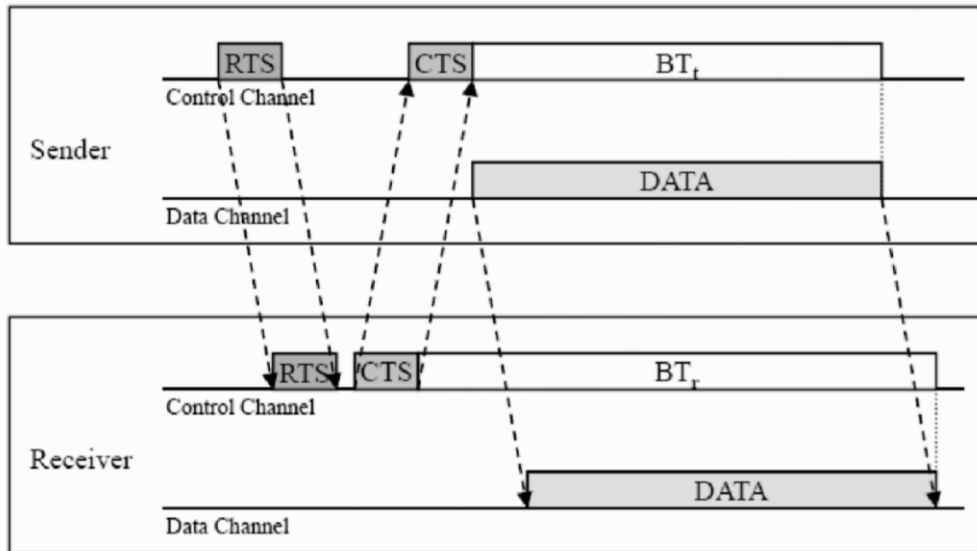


Figure 3.6 Packet transmission in DBTMA 3.5.2 Receiver-initiated protocols:

a) RECEIVER INITIATED-BUSY TONE MULTIPLE ACCESS PROTOCOL (RI-BTMA):

In this RI-BTMA similar to BTMA, the available bandwidth is divided into two channels: a data channel for transmitting data packets and a control channel. The control channel is used by a node to transmit the busy tone signal. A node can transmit on the data channel only if it finds the busy tone to be absent on the control channel.

The data packet is divided into two portions: a preamble and the actual data packet. The preamble carries the identification of the intended destination node. Both the data channel and the control channel are slotted, with each slot equal to the length of the preamble. Data transmission consists of two steps. First, the preamble needs to be transmitted by the sender. Once the receiver node acknowledges the reception of this preamble by transmitting the busy tone signal on the control channel, the actual data packet is transmitted. A sender node that needs to transmit a data packet first waits for a

free slot, that is, a slot in which the busy tone signal is absent on the control channel. Once it finds such a slot, it transmits the preamble packet on the data channel. If the destination node receives this preamble packet correctly without any error, it transmits the busy tone on the control channel. It continues transmitting the busy tone signal as long as it is receiving data from the sender. If preamble transmission fails, the receiver does not acknowledge with the busy tone, and the sender node waits for the next free slot and tries again. The busy tone serves two purposes. First, it acknowledges the sender about the successful reception of the preamble. Second, it informs the nearby hidden nodes about the impending transmission so that they do not transmit at the same time.

There are two types of RI-BTMA protocols: the basic protocol and the controlled protocol. In the basic protocol, nodes do not have backlog buffers to store data packets. Hence packets that suffer collisions cannot be retransmitted. Also, when the network load increases, packets cannot be queued at the nodes. This protocol would work only when the network load is not high; when network load starts increasing, the protocol becomes unstable.

The controlled protocol overcomes this problem. This protocol is the same as the basic protocol, the only difference being the availability of backlog buffers at nodes. Therefore, packets that suffer collisions, and those that are generated during busy slots, can be queued at nodes. A node is said to be in the backlogged mode if its backlog buffer is non-empty. When a node in the backlogged mode receives a packet from its higher layers, the packet is put into the buffer and transmitted later.

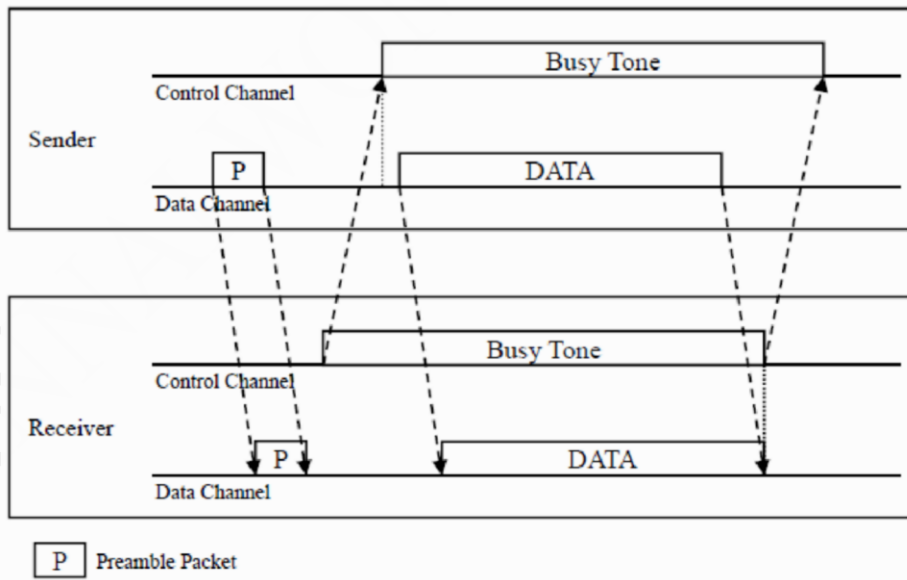


Figure 3.7 Packet transmissions in RI-BTMA b)

MACA-By Invitation

MACA-by invitation (MACA-BI) is a receiver-initiated MAC protocol. It reduces the number of control packets used in the MACA protocol. MACA, which is a sender-initiated protocol, uses the three-way handshake mechanism, where first the RTS and CTS control packets are exchanged, followed by the actual DATA packet transmission. MACA-BI eliminates the need for the RTS packet. In MACA-BI the receiver node initiates data transmission by transmitting a Ready To Receive (RTR) control packet to the sender. If it is ready to transmit, the sender node responds by sending a DATA packet. Thus data transmission in MACA-BI occurs through a two-way handshake mechanism. The efficiency of the MACA-BI scheme is mainly dependent on the ability of the receiver node to predict accurately the arrival rates of traffic at the sender nodes.

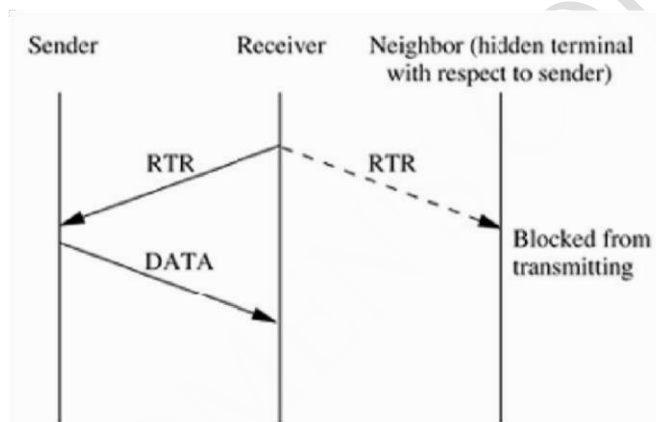


Figure 3.8 Packet transmissions in MACA-By

c) Media Access with Reduced Handshake (MARCH):

The media access with reduced handshake protocol (MARCH) is a receiver-initiated protocol. MARCH, unlike MACA-BI, does not require any traffic prediction mechanism. The protocol exploits the broadcast nature of traffic from omnidirectional antennas to reduce the number of handshakes involved in data transmission. In MACA, the RTS-CTS control packets exchange takes place before the transmission of every data packet. But in MARCH, the RTS packet is used only for the first packet of the stream. From the second packet onward, only the CTS packet is used. A node obtains information about data packet arrivals at its neighboring nodes by overhearing the CTS packets transmitted by them. It then sends a CTS packet to the concerned neighbor node for relaying data from that node.

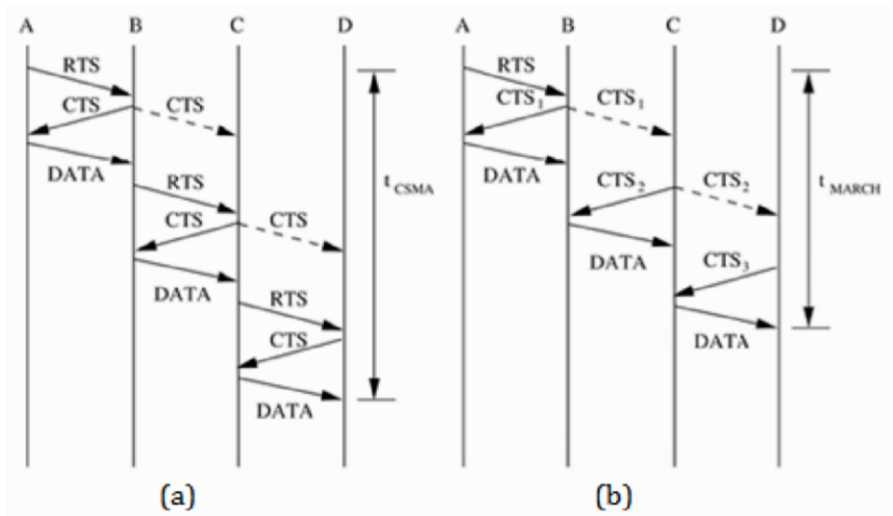


Figure 3.9 Handshake mechanism in (a) MACA and (b) MARCH
3.6 CONTENTION-BASED PROTOCOLS WITH RESERVATION MECHANISMS:

These protocols are contention-based, contention occurs only during the resource (bandwidth) reservation phase. Once the bandwidth is reserved, the node gets exclusive access to the reserved bandwidth. Hence, QoS support can be provided for real-time traffic.

3.6.1 Distributed Packet Reservation Multiple Access Protocol

The Distributed Packet Reservation Multiple Access protocol (D-PRMA) extends the earlier centralized Packet Reservation Multiple Access (PRMA) scheme into a distributed scheme that can be used in ad hoc wireless networks. PRMA was proposed for voice support in a wireless LAN with a base station, where the base station serves as the fixed entity for the MAC operation. D-PRMA extends this protocol for providing voice support in ad hoc wireless networks.

D-PRMA is a TDMA-based scheme. The channel is divided into fixed- and equal-sized frames along the time axis. Each frame is composed of s slots, and each slot consists of m mini-slots. Each mini-slot can be further divided into two control fields, RTS/BI and CTS/BI (BI stands for Busy Indication). These control fields are used for slot reservation and for overcoming the hidden terminal problem. All nodes having packets ready for transmission contend for the first mini-slot of each slot. The remaining $(m - 1)$ mini-slots are granted to the node that wins the contention. Also, the same slot in each subsequent frame can be reserved for this winning terminal until it completes its packet transmission session.

If no node wins the first mini-slot, then the remaining mini-slots are continuously used for contention, until a contending node wins any mini-slot. Within a reserved slot, communication between the source and receiver nodes takes place by means of either Time Division Duplexing (TDD) or Frequency Division Duplexing (FDD). Any node that wants to transmit packets has to first reserve slots, if they have not been reserved already. A certain period at the beginning of each mini-slot is reserved for carrier sensing. If a sender node detects the channel to be idle at the beginning of a slot (minislot 1), it transmits an RTS packet (slot reservation request) to the intended destination through the RTS/BI part of the current mini-slot. On successfully receiving this RTS packet, the receiver node responds by sending a CTS packet through the CTS/BI of the same mini-slot. If the sender node receives this CTS successfully, then it gets the reservation for the current slot and can use the remaining mini-slots, that is, mini-slots 2 to m . Otherwise, it continues the contention process through the subsequent mini-slots of the same slot.

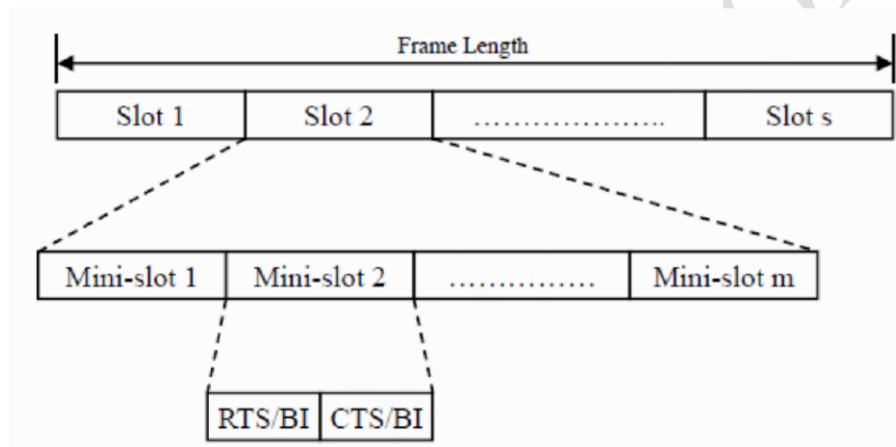


Figure 3.10 Frame structure in D-PRMA

To prioritize voice terminals over data terminals, the Voice terminals starts contenting from minislot 1 with probability $p = 1$ while data terminals can start such content with $p < 1$. Both voice and data terminals can content through the extra $(m - 1)$ mini-slots with probability $p < 1$. Only the winner of a voice terminal can reserve the same slot in each subsequent frame until the end of packet transmission while the winner of a data terminal can only use one slot.

3.6.2 Collision Avoidance Time Allocation Protocol (CATA):

The Collision Avoidance Time Allocation protocol (CATA) is based on dynamic topology dependent transmission scheduling. Nodes contend for and reserve time slots by means of a distributed reservation and handshake mechanism. CATA supports broadcast, unicast,

and multicast transmissions simultaneously. The operation of CATA is based on two basic principles:

1. The receiver(s) of a flow must inform the potential source nodes about the reserved slot on which it is currently receiving packets. Similarly, the source node must inform the potential destination node(s) about interferences in the slot.
2. Usage of negative acknowledgments for reservation requests, and control packet transmissions at the beginning of each slot, for distributing slot reservation information to senders of broadcast or multicast sessions.

Time is divided into equal-sized frames, and each frame consists of S slots. Each slot is further divided into five mini-slots. The first four mini-slots are used for transmitting control packets and are called control mini-slots (CMS1, CMS2, CMS3, and CMS4). The fifth and last Minislot, called data mini-slot (DMS), is meant for data transmission. The data mini-slot is much longer than the control mini-slots as the control packets are much smaller in size compared to data packets.

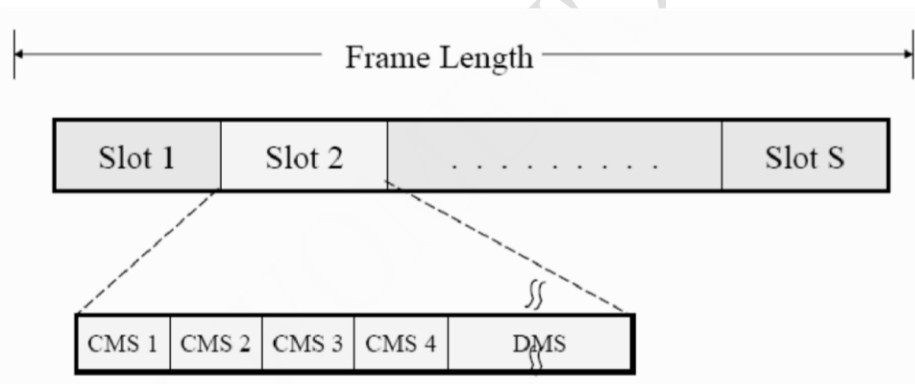


Figure 3.11 Frame structure in CATA

Each node receives data during the DMS of current slot transmits an SR in CMS1. Every node that transmits data during the DMS of current slot transmits an RTS in CMS2.

CMS3 and CMS4 are used as follows:

- a. The sender of an intend reservation, if it senses the channel is idle in CMS1, transmits an RTS in CMS2.
- b. Then the receiver transmits a CTS in CMS3
- c. If the reservation was successful the data can transmit in current slot and the same slot in subsequent frames.
- d. Once the reservation was successfully, in the next slot both the sender and receiver do not transmit anything during CMS3 and during CMS4 the sender transmits a NTS.

If a node receives an RTS for broadcast or multicast during CMS2 or it finds the channel to be free during CMS2, it remains idle during CMS3 and CMS4. Otherwise it sends a NTS packet during CMS4. A potential multicast or broadcast source node that receives the NTS packet or detecting noise during CMS4, understands that its reservation is failed. If it find the channel is free in CMS4, which implies its reservation was successful. CATA works well with simple single-channel halfduplex radios.

3.6.3 Hop Reservation Multiple Access Protocol:

HRMA is a multi-channel MAC protocol, based on half-duplex very slow frequency hopping spread spectrum (FHSS) radios. Each time slot is assigned a separate frequency channel

Assume L frequency channels, f_0 - dedicated synchronized channel frequency.

The remaining $L-1$ frequencies $L-1$ are divided into $M = \frac{L-1}{2}$ frequency pairs denoted by

$(f_i, f_{i^*}), i=1,2,3,4,\dots,M$, Hop reservation (HR),

RTS, CTS, DATA : f_i and ACK f_{i^*}

All idle nodes hop to the synchronizing frequency f_0 and exchange synchronization information. Synchronizing slot: used to identify the beginning of a frequency hop and the frequency to be used in the immediately following hop. Any two nodes from two disconnected networks have at least two overlapping time period of length μ_s on the frequency f_0 .

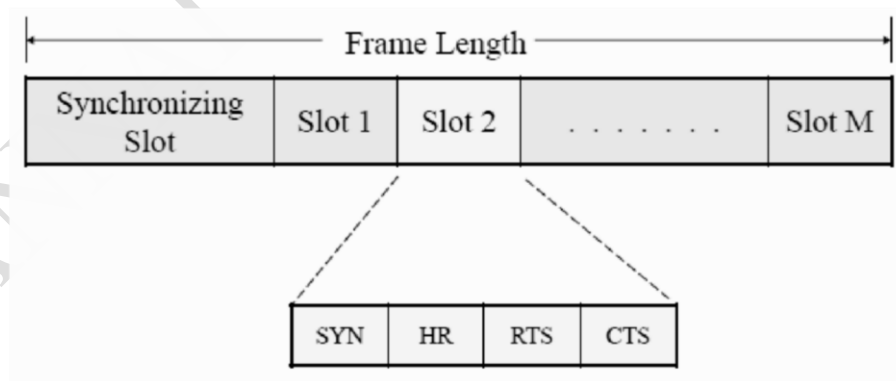


Figure 3.12 Frame format in HRMA

If μ is the length of each slot and μ_s is the length of the synchronization period on each slot, then the dwell time of f_0 is $\mu + \mu_s$.

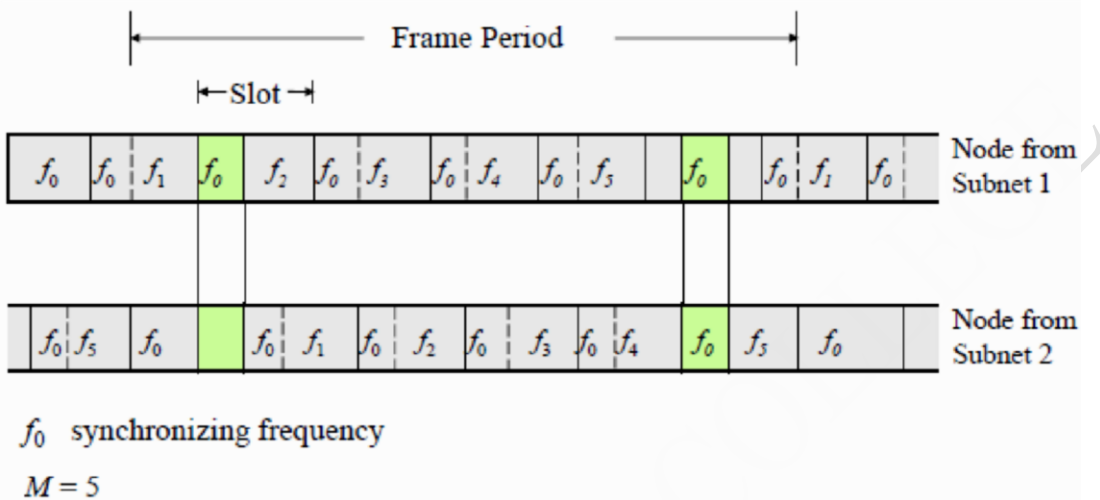


Figure 3.13 Merging of Subnets

A node ready to transmit data, it senses the HR period of the current slot, then if the channel is idle during HR period, it transmits an RTS during RTS period and waits for CTS during CTS period. If the channel is busy during HR period, it backs off for a randomly multiple slots. Suppose the sender needs to transmit data across multiple frames, it informs the receiver through the header of the data packet. The receiver node transmits an HR packet during the HR period of the same slot in next frame to inform its neighbors. The sender receiving the HR packet, it sends an RTS during the RTS period and jams other RTS packets. Then Both sender and receiver remain silent during the CTS period.

3.6.4 FPRP: Five-Phase Reservation Protocol

A single-channel TDMA based broadcast scheduling protocol. Nodes use a contention mechanism in order to acquire time slots. The protocol assumes the availability of global time at all nodes. Time is divided into frames: reservation frame (RF) and information frame (IF). Each RF has N reservation slots (RS) and each IF has N information slots (IS). Each RS is composed of M reservation cycles (RCs). With each RC, a five-phase dialog takes place. Corresponding to IS, each node would be in one of the following three states: transmit (T), receive (R), and blocked (B).

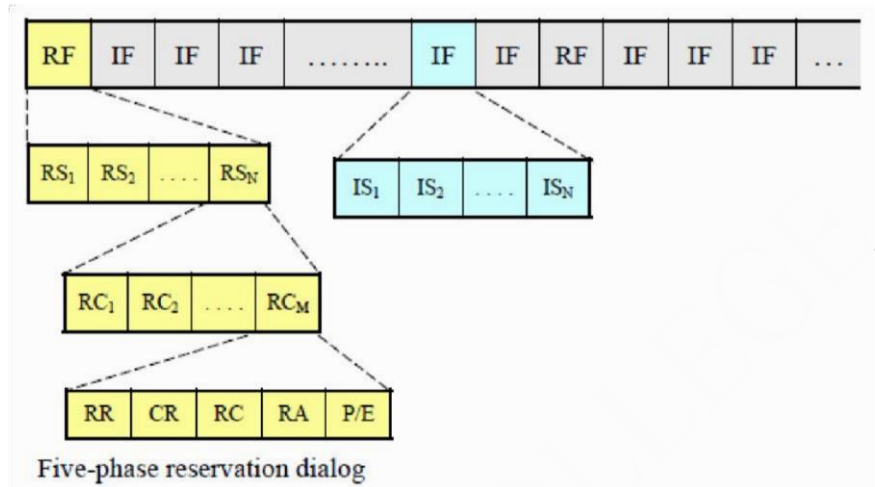


Figure 3.14 Frame structure of FPRP

The reservation takes five phases:

1. **Reservation request:** Send reservation request (RR) packet to dest.
2. **Collision report:** If a collision is detected by any node, that node broadcasts a CR packet
3. **Reservation confirmation:** A source node won the contention will send a RC packet to destination node if it does not receive any CR message in the previous phase
4. **Reservation acknowledgment:** Destination node acknowledge reception of RC by sending back RA message to source
5. **Packing and elimination:** Use packing packet and elimination packet.

Example:

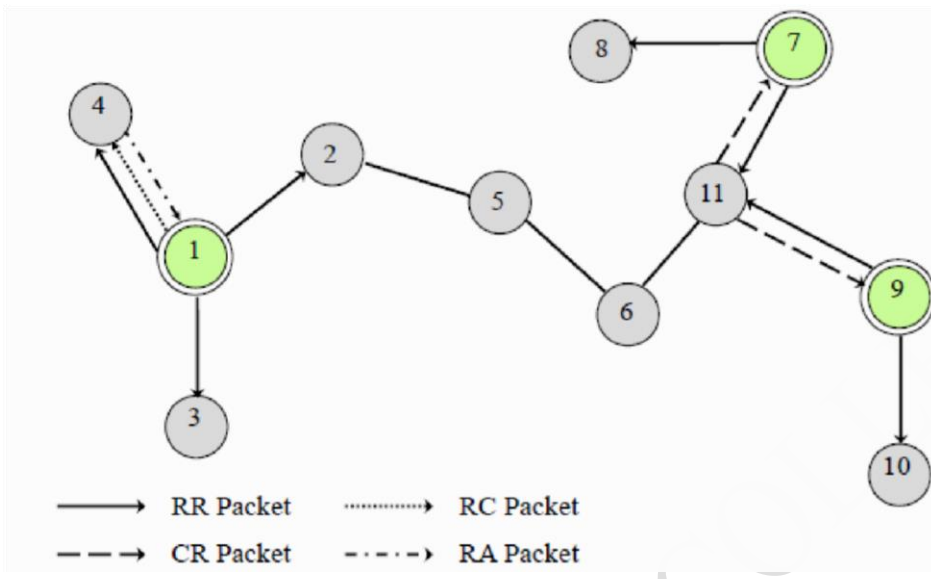


Figure 3.15 Example of FFRP

Here nodes 1, 7, and 9 have packets ready to be transmitted to nodes 4, 8, and 10, respectively. During the reservation request phase, all three nodes transmit RR packets. Since no other node in the two-hop neighborhood of node 1 transmits simultaneously, node 1 does not receive any CR message in the collision report phase. So node 1 transmits an RC message in the next phase, for which node 4 sends back an RA message, and the reservation is established. Node 7 and node 9 both transmit the RR packet in the reservation request phase. Here node 9 is within two hops from node 7. So if both nodes 7 and 9 transmit simultaneously, their RR packets collide at common neighbor node 11. Node 11 sends a CR packet which is heard by nodes 7 and 9. On receiving the CR packet, nodes 7 and 9 stop contending for the current slot.

3.6.5 MACA/PR: MACA with Piggy- Backed Reservation

MACA/PR is used to provide real time traffic support. The main components: a MAC protocol (MACAW + non persistent CSMA), a reservation protocol, and a QoS routing protocol. Each node maintains a reservation table (RT) that records all the reserved transmit and receive slots/windows of all nodes. Non-real time packet: wait for a free slot in the RT + random time => RTS => CTS => DATA => ACK. Real time packet transmit real time packets at certain regular intervals (say CYCLE).

RTS=>CTS=>DATA (carry reservation info for next data) => ACK=>... =>DATA

(carry reservation info)=>ACK, Hear DATA and ACK: update their reservation table.

The ACK packet serves to renew the reservation, in addition to recovering from the packet loss.

Reservation fail: fail to receive ACK packets for a certain number of DATA packets.

For maintaining consistent information regarding free slots, Periodic exchange of reservation tables. Best effort and real time packet transmissions can be interleaved at nodes. When a new node joins: receive reservation tables from each of its neighbors and learns about the reservations made in the network. QoS Routing protocol: DSDV (destination sequenced distance vector). MACA/PR does not require global synchronization among nodes. Drawback is possibility of many fragmented free slots not being used at all.

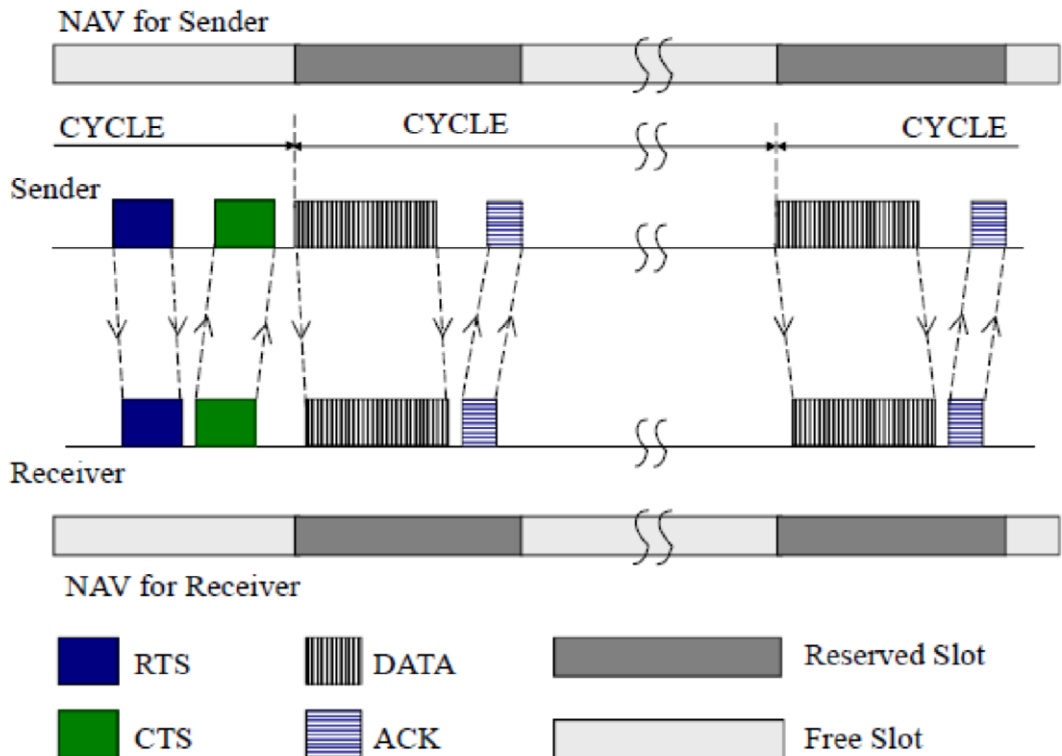


Figure 3.16 Packet transmission in MACA/PR

3.6.6 RTMAC: Real Time Medium Access Control Protocol

The real-time medium access control protocol (RTMAC) provides a bandwidth reservation mechanism for supporting real-time traffic in ad hoc wireless networks. RTMAC consists of two components, a MAC layer protocol and a QoS routing protocol. The MAC layer protocol is a realtime extension of the IEEE 802.11 DCF. The QoS routing protocol is responsible for end-to-end reservation and release of bandwidth resources.

The MAC layer protocol has two parts: a medium-access protocol for best-effort traffic and a reservation protocol for real-time traffic. A separate set of control packets, consisting of ResvRTS, ResvRTSResvCTS, and ResvACK, is used for effecting bandwidth reservation for realtime packets. RTS, CTS, and ACK control packets are

used for transmitting best-effort packets. In order to give higher priority for real-time packets, the wait time for transmitting a ResvRTS packet is reduced to half of DCF inter-frame space (DIFS), which is the wait time used for best-effort packets.

Time is divided into super-frames. As can be seen from Figure 6.24, the super-frame for each node may not strictly align with the other nodes. Bandwidth reservations can be made by a node by reserving variable-length time slots on super-frames, which are sufficient enough to carry the traffic generated by the node. Each super-frame consists of a number of reservation-slots (resv-slots). The time duration of each resv-slot is twice the maximum propagation delay. Data transmission normally requires a block of resvslots. A node that needs to transmit real-time packets first reserves a set of resv-slots. The set of resv-slots reserved by a node for a connection on a superframe is called a connection-slot.

A node that has made reservations on the current super-frame makes use of the same connection-slot in the successive super-frames for transmitting packets. Each node maintains a reservation table containing information such as the sender id, receiver id, and starting and ending times of reservations that are currently active within its direct transmission range.

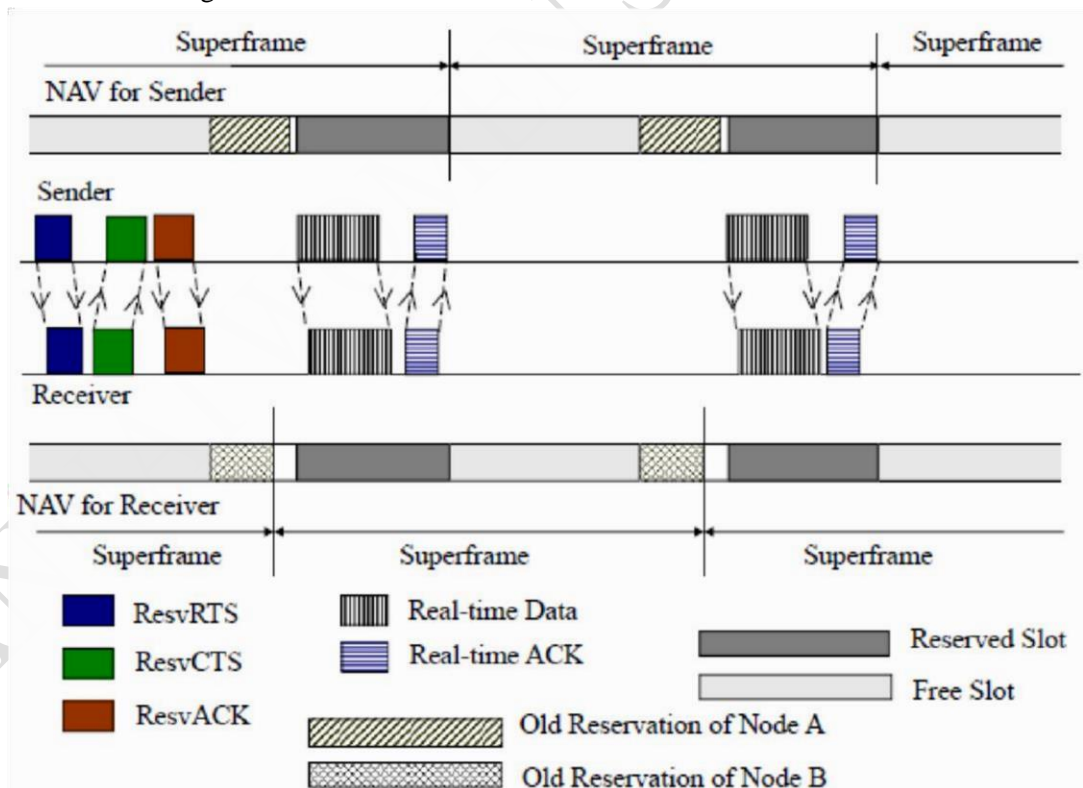


Figure 3.17 Reservation mechanism in RTMAC

3.7 CONTENTION-BASED MAC PROTOCOLS WITH SCHEDULING MECHANISMS:

Protocols in this category focus on packet scheduling at the nodes and transmission scheduling of the nodes. The factors that affects scheduling decisions are Delay targets of packets, Traffic load at nodes and Battery power.

3.7.1 Distributed Priority Scheduling and Medium Access in Ad Hoc Networks:

Distributed priority scheduling and medium access in Ad Hoc Networks present two mechanisms for providing quality of service (QoS). They are Distributed priority scheduling (DPS) – Piggy-backs the priority tag of a node's current and head-of-line packets to the control and data packets and Multi-hop coordination – Extends the DPS scheme to carry out scheduling over multihop paths.

The distributed priority scheduling scheme (DPS) is based on the IEEE 802.11 distributed coordination function. DPS uses the same basic RTS-CTS-DATA-ACK packet exchange mechanism. The RTS packet transmitted by a ready node carries the priority tag/priority index for the current DATA packet to be transmitted. The priority tag can be the delay target for the DATA packet. On receiving the RTS packet, the intended receiver node responds with a CTS packet. The receiver node copies the priority tag from the received RTS packet and piggybacks it along with the source node id, on the CTS packet. Neighbor nodes receiving the RTS or CTS packets (including the hidden nodes) retrieve the piggy-backed priority tag information and make a corresponding entry for the packet to be transmitted, in their scheduling tables (STs).

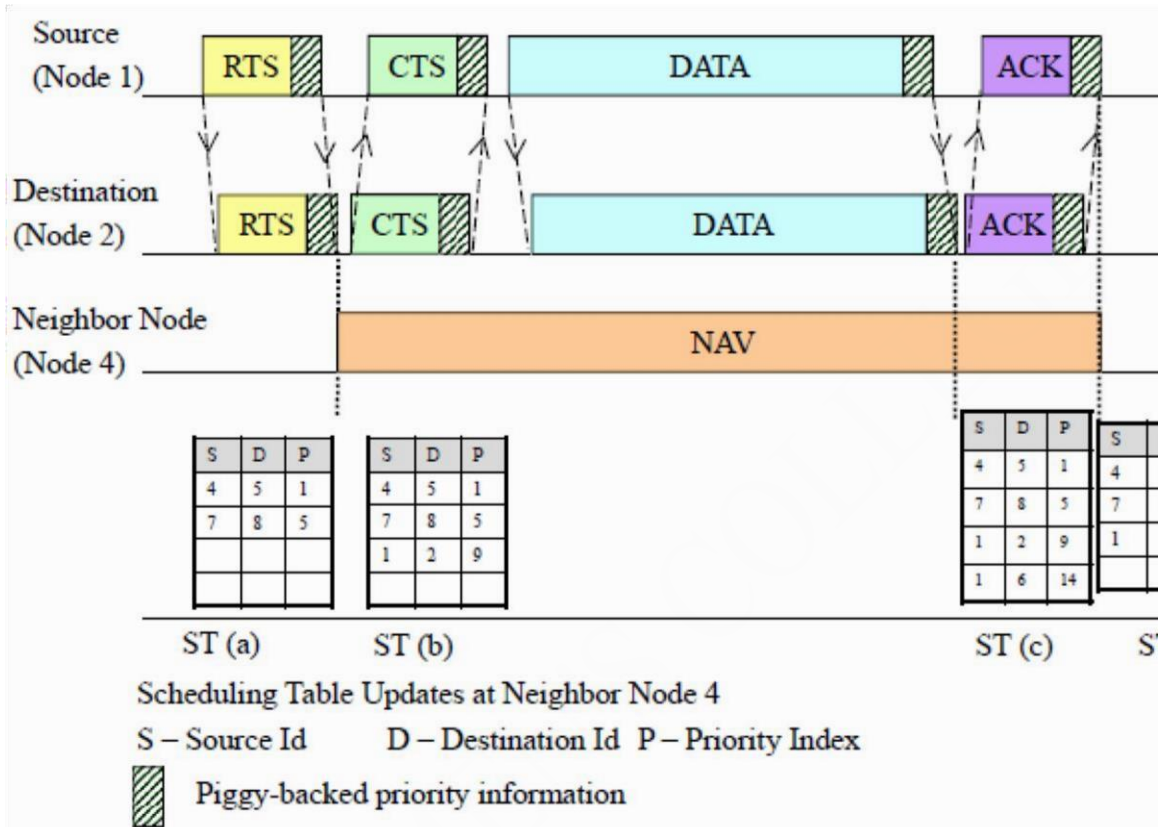


Figure 3.18 Piggy-backing and scheduling table update mechanism in DPS

3.7.2 Distributed Wireless Ordering Protocol (DWOP)

The distributed wireless ordering protocol (DWOP) consists of a media access scheme along with a scheduling mechanism. It is based on the distributed priority scheduling scheme. DWOP ensures that packets access the medium according to the order specified by an ideal reference scheduler such as first-in-first-out (FIFO), virtual clock, or earliest deadline first. In this discussion, FIFO is chosen as the reference scheduler. In FIFO, packet priority indices are set to the arrival times of packets. Similar to DPS, control packets are used in DWOP to piggy-back priority information regarding head-of-line packets of nodes. Example:

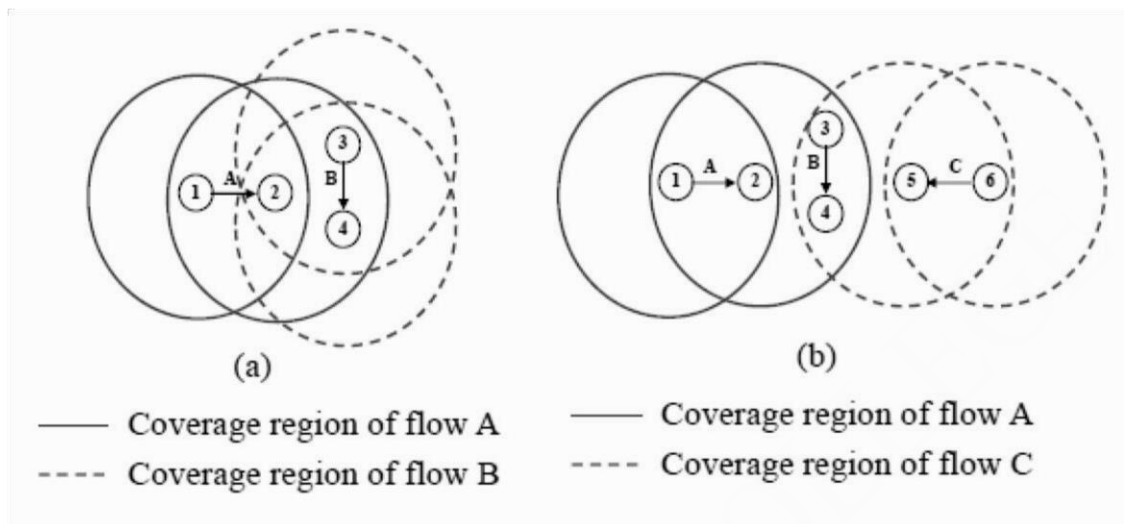


Figure 3.19 (a) Information asymmetry. (b) Perceived collisions

1. Information asymmetry: A transmitting node might not be aware of the arrival times of packets queued at another node which is not within its transmission range. The Solution is a receiver find that the sender is transmitting out of order, an out-of-order notification is piggy-backed by the receiver on the control packet (CTS/ACK).
2. Perceived collisions: The ACK packet collides at the node, the corresponding entry in the ST will never be removed. The Solution is when a node observes that its rank remains fixed while packets whose PR are below the priority of its packet are being transmitted, it deletes the oldest entry from its ST.

In summary, DWOP tries to ensure that packets get access to the channel according to the order defined by a reference scheduler. The above discussion was with respect to the FIFO scheduler. Though the actual schedule deviates from the ideal FIFO schedule due to information asymmetry and stale information in STs, the receiver participation and the stale entry elimination mechanisms try to keep the actual schedule as close as possible to the ideal schedule.

3.7.3 Distributed Laxity-Based Priority Scheduling Scheme:

The distributed laxity-based priority scheduling (DLPS) scheme is a packet scheduling scheme, where scheduling decisions are made taking into consideration the states of neighboring nodes and the feedback from destination nodes regarding packet losses. Packets are reordered based on their uniform laxity budgets (ULBs) and the packet delivery ratios of the flows to which they belong.

Each node maintains two tables: scheduling table (ST) and packet delivery ratio table (PDT). The ST contains information about packets to be transmitted by the node and packets overheard by the node, sorted according to their priority index values. Priority index expresses the priority of a packet. The lower the priority index, the higher the packet's priority. The PDT contains the count of packets transmitted and the count of acknowledgment (ACK) packets received for every flow passing through the node. This information is used for calculating current packet delivery ratio of flows.

3.8 MAC PROTOCOLS THAT USE DIRECTIONAL ANTENNAS:

MAC protocols that use directional antennas for transmissions have several advantages over those that use omnidirectional transmissions. The advantages include reduced signal interference, increase in the system throughput, and improved channel reuse that leads to an increase in the overall capacity of the channel.

A directional antenna or beam antenna is an antenna which radiates or receives greater power in specific directions allowing for increased performance and reduced interference from unwanted sources

3.8.1 MAC Protocol Using Directional Antennas

The MAC protocol for mobile ad hoc networks using directional antennas makes use to improve the throughput in ad hoc wireless networks. The mobile nodes do not have any location information by means of which the direction of the receiver and sender nodes could be determined. The protocol makes use of an RTS/CTS exchange mechanism, which is similar to the one used in MACA. The nodes use directional antennas for transmitting and receiving data packets, thereby reducing their interference to other neighbor nodes. This leads to an increase in the throughput of the system. Each node is assumed to have only one radio transceiver, which can transmit and receive only one packet at any given time. The transceiver is assumed to be equipped with M directional antennas, each antenna having a conical radiation pattern, spanning an angle of $2\pi/M$ radians

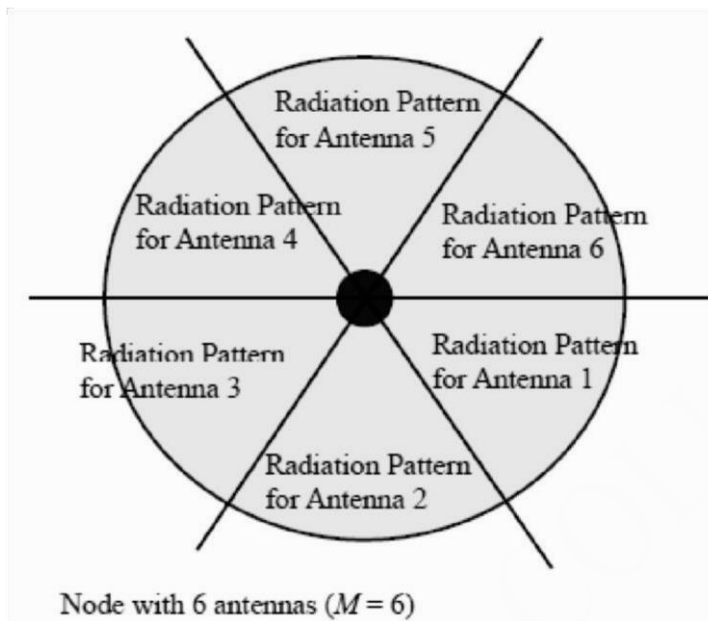


Figure 3.20 Radiation patterns of directional antennas

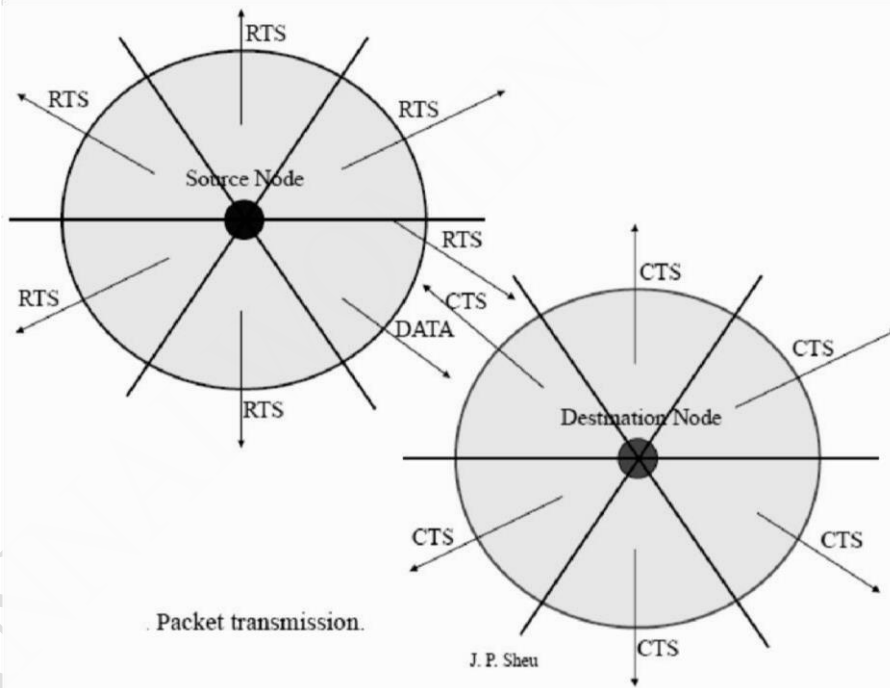


Figure 3.21 Packet Transmission in directional antennas

3.8.2 Directional Busy Tone-Based MAC Protocol:

The directional busy tone-based MAC protocol adapts the DBTMA protocol for use with directional antennas. It uses directional antennas for transmitting the RTS, CTS, and data frames, as well as the busy tones. By doing so, collisions are reduced significantly. Also, spatial reuse of the channel improves, thereby increasing the capacity of the channel. Each node has a directional antenna which consists of N antenna elements, each covering a fixed sector spanning an angle of $(360/N)$ degrees. For a unicast transmission, only a single antenna element is used. For broadcast transmission, all the N antenna elements transmit simultaneously.

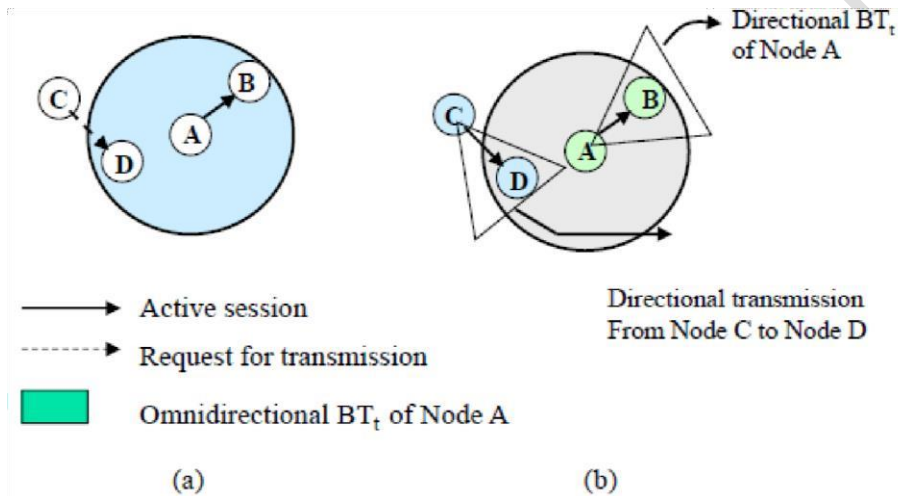


Figure 3.22 Directional DBTMA(Example-1)

When a node is idle (not transmitting packets), all antenna elements of the node keep sensing the channel. The node is assumed to be capable of identifying the antenna element on which the incoming signal is received with maximum power. Therefore, while receiving, exactly one antenna element collects the signals. In an ad hoc wireless network, nodes may be mobile most of the time. It is assumed that the orientation of sectors of each antenna element remains fixed. The protocol uses the same two busy tones BT_t and BT_r used in the DBTMA protocol. A node that receives a data packet for transmission first transmits an RTS destined to the intended receiver in all directions (omnidirectional transmission). On receiving this RTS, the receiver node determines the antenna element on which the RTS is received with maximum gain. This will observe in figure 3.22(b). This protocol is not guaranteed to be collision free see Fig. 3.23(b).

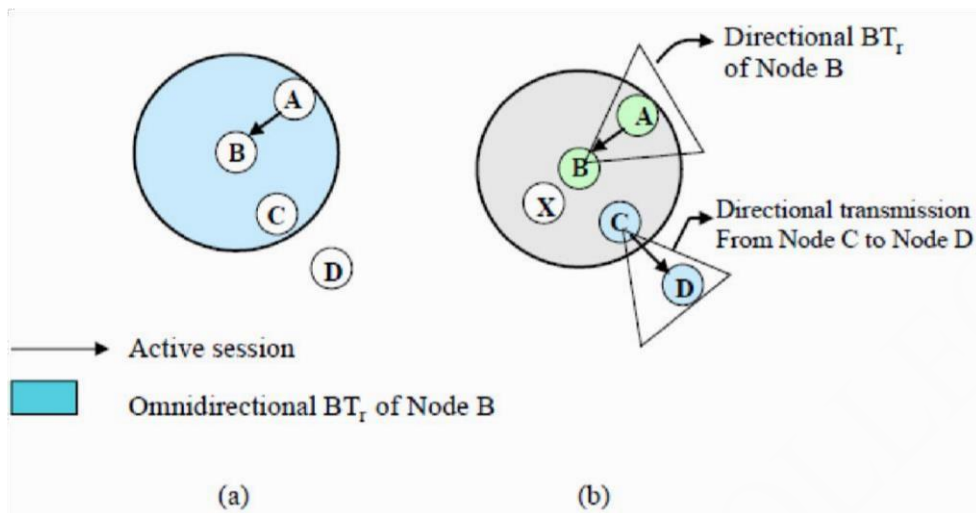


Figure 3.23 Directional DBTMA(Example-2)

3.8.3 Directional MAC Protocols for Ad-Hoc Wireless Networks:

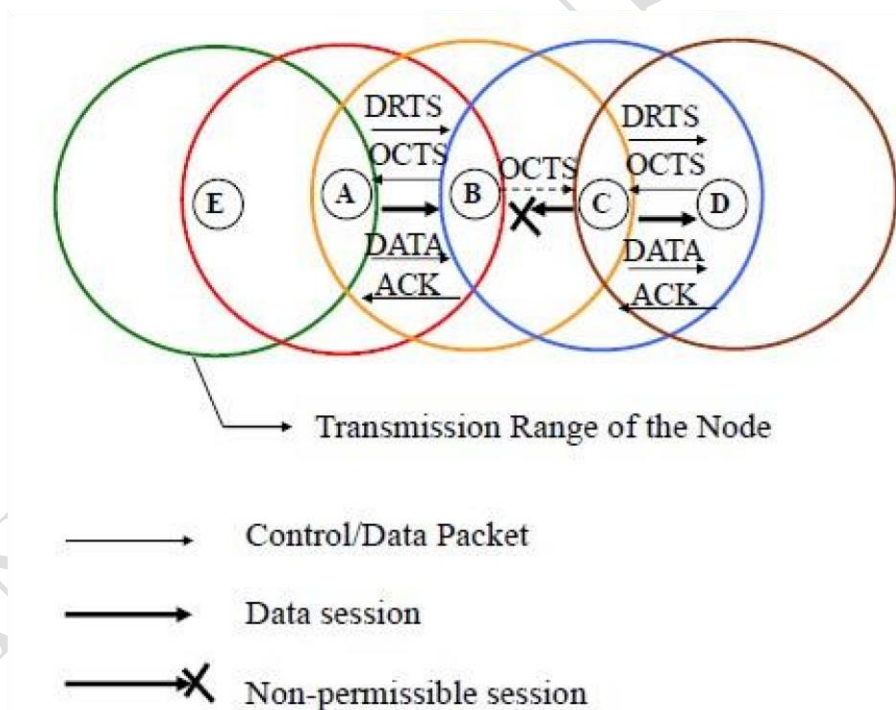


Figure 3.24 Operation of D-MAC Protocol

1. D-MAC: assume each node knows about the location of neighbors
2. In the first directional MAC scheme (DMAC-1)
 - a. Directional RTS (DRTS) → Omni-directional CTS (OCTS) → Directional DATA (DDATA) → Directional ACK (DACK).
 - b. May increase the probability of control packet collisions
 - c. See Figure 3.24 (if node E send a packet to node A, it will collide the OCTS or DACK)
3. In the second directional MAC scheme (DMAC-2)
 - a. Both the Directional RTS (DRTS) and Omni-directional RTS (ORTS) transmissions are used.
 - b. Reduced control packet collisions
4. Rules for using DRTS and ORTS:
 - a. ORTS: None of the directional antennas are blocked
 - b. DRTS: Otherwise.
 - c. Another packet called directional wait-to-send (DWTS) is used in this scheme (See Figure 3.25)

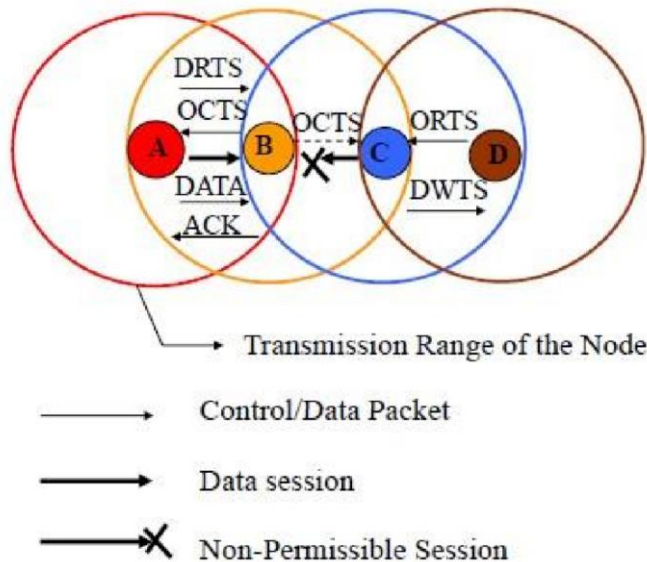


Figure 3.25 Operation of D-MAC Protocol

3.9 OTHER MAC PROTOCOLS:

There are several other MAC protocols that do not strictly fall under the three contention based protocol categories.

3.9.1 Multichannel MAC Protocol:

The multichannel MAC protocol (MMAC) [24] uses multiple channels for data transmission. There is no dedicated control channel. N channels that have enough spectral separation between each other are available for data transmission. Each node

maintains a data structure called Preferable Channel List (PCL). The usage of the channels within the transmission range of the node is maintained in the PCL. Based on their usage, channels can be classified into three types.

1. *High preference channel (HIGH)*: The channel has been selected by the current node and is being used by the node in the current beacon interval (beacon interval mechanism will be explained later). Since a node has only one transceiver, there can be only one HIGH channel at a time.
2. *Medium preference channel (MID)*: A channel which is free and is not being currently used in the transmission range of the node is said to be a medium preference channel. If there is no HIGH channel available, a MID channel would get the next preference.
3. *Low preference channel (LOW)*: Such a channel is already being used in the transmission range of the node by other neighboring nodes. A counter is associated with each LOW state channel. For each LOW state channel, the count of source-destination pairs which have chosen the channel for data transmission in the current beacon interval is maintained.

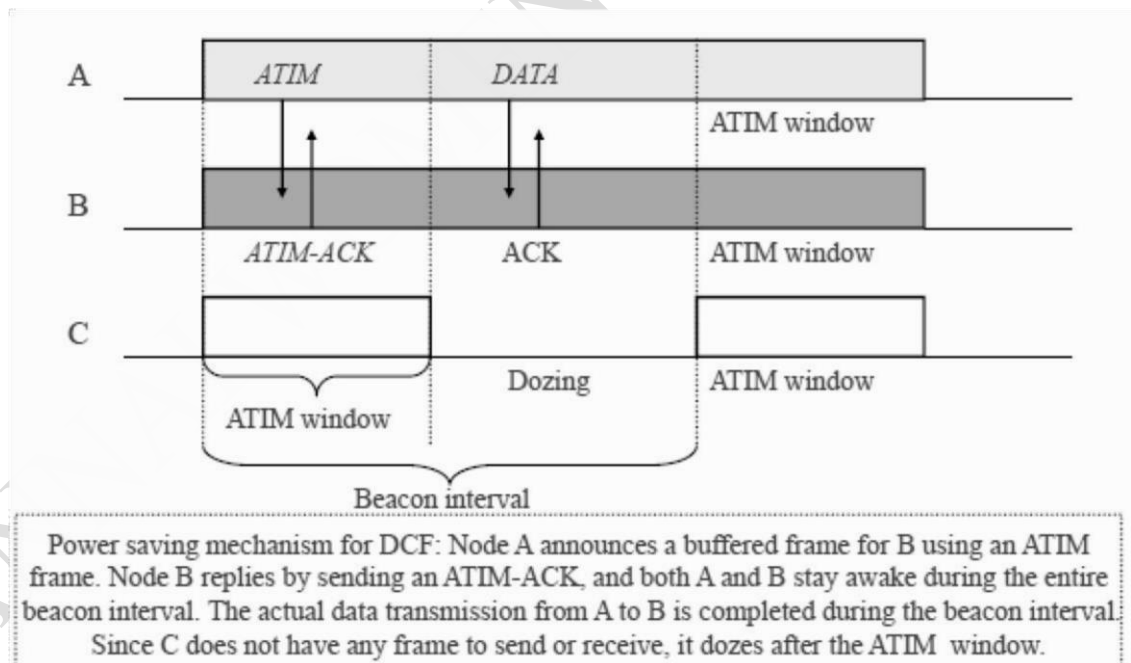


Figure 3.26 ATIM Window

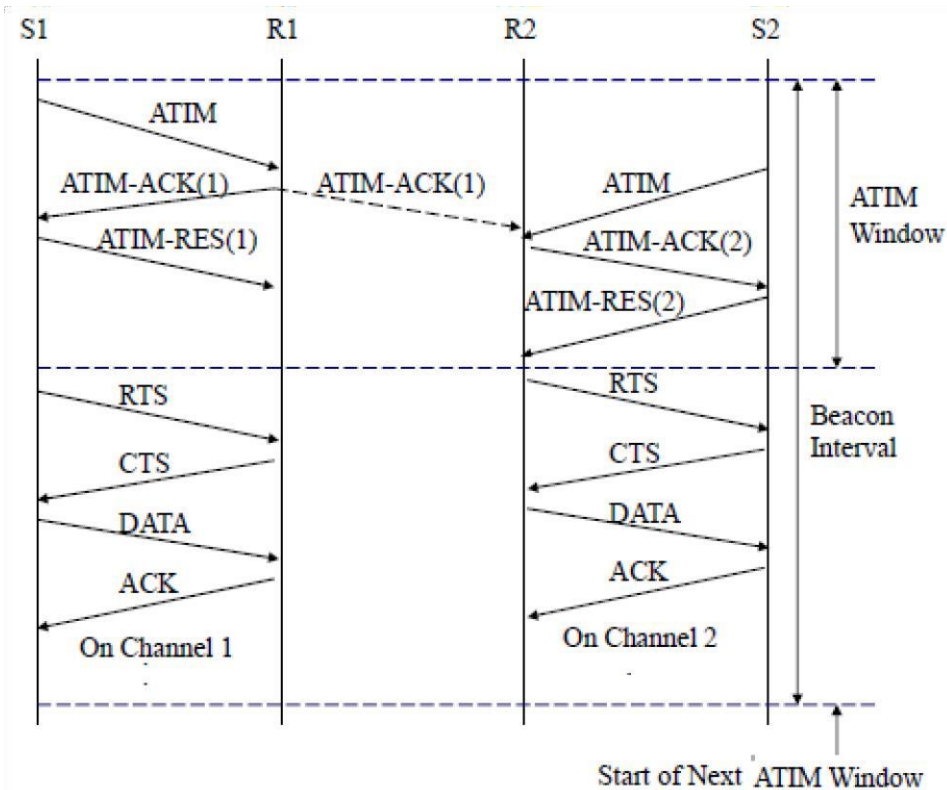


Figure 3.27 Operation of MMAC Protocol

Time is divided into beacon intervals and every node is synchronized by periodic beacon transmissions. ATIM messages such as ATIM, ATIM-ACK (ATIMacknowledgment), and ATIM-RES (ATIM-reservation) are used for this negotiation. The exchange of ATIM messages takes place on a particular channel called the default channel. The default channel is one of the multiple available channels. This channel is used for sending DATA packets outside the ATIM window, like any other channel. The ATIM message carries the PCL of the transmitting node. The destination node, upon receiving the packet, uses the PCL carried on the packet and its own PCL to select a channel. The ATIM packets themselves may be lost due to collisions; in order to prevent this, each node waits for a randomly chosen back-off period before transmitting the ATIM packet.

Channel selection mechanism:

1. If a HIGH state channel exists in node R's PCL then that channel is selected.
2. Else if there exists a HIGH state channel in the PCL of node S then this channel is selected

3. Else if there exists a common MID state channel in the PCLs of both node S and node R then that channel is selected.
4. Else if there exists a MID state at only one of the two nodes then that channel is selected.
5. If all channels in both PCLs are in LOW state the channel with the least count is selected.

MMAC uses simple hardware. It requires only a single transceiver. It does not have any dedicated control channel. The throughput of MMAC is higher than that of IEEE 802.11 when the network load is high. This higher throughput is in spite of the fact that in MMAC only a single transceiver is used at each node. Unlike other protocols, the packet size in MMAC need not be increased in order to take advantage of the presence of an increased number of channels.

3.9.2 Multichannel CSMAMAC Protocol:

In the multichannel CSMA MAC protocol (MCSMA), the available bandwidth is divided into several channels. A node with a packet to be transmitted selects an idle channel randomly. The protocol also employs the notion of soft channel reservation, where preference is given to the channel that was used for the previous successful transmission. Though the principle used in MCSMA is similar to the frequency division multiple access (FDMA) schemes used in cellular networks, the major difference here is that there is no centralized infrastructure available, and channel assignment is done in a distributed fashion using carrier-sensing.

The total available bandwidth is divided into N non-overlapping channels. Where N is independent of the number of hosts in the network, each having a bandwidth of (W/N) , where W is the total bandwidth available for communication. The channels may be created in the frequency domain (FDMA) or in the code domain (CDMA). Since global synchronization between nodes is not available in ad hoc wireless networks, channel division in the time domain (TDMA) is not used.

When the number of channels N is sufficiently large, each node tends to reserve a channel for itself. This is because a node prefers the channel used in its last successful transmission for its next transmission also. This reduces the probability of two contending nodes choosing the same channel for transmission. Even at high traffic loads, due to the tendency of every node to choose a reserved channel for itself, the chances of collisions are greatly reduced. The number of channels into which the available

bandwidth is split is a very important factor affecting the performance of the protocol. If the number of channels is very large, then the protocol results in very high packet transmission times.

3.9.3 Power Control MAC Protocol for Ad Hoc Networks:

The power control MAC protocol (PCM) allows nodes to vary their transmission power levels on a per-packet basis. In the BASIC scheme, the RTS and CTS packets are transmitted with maximum power P_{MAX} . The RTS-CTS handshake is used for deciding upon the transmission power for the subsequent DATA and ACK packet transmissions. This can be done using two methods. In the first method, source node A transmits the RTS with maximum power P_{MAX} . This RTS is received at the receiver with signal level P_R . The receiver node B can calculate the minimum required transmission power level $P_{DESIRED}$ for the DATA packet, based on the received power level P_R , the transmitted power level P_{MAX} , and the noise level at receiver B. Node B then specifies this

$P_{DESIRED}$ in the CTS packet it transmits to node A.

In the second method, when the receiver node B receives an RTS packet, it responds with a CTS packet at the usual maximum power level P_{MAX} . When the source node receives this CTS packet, it calculates $P_{DESIRED}$ based on the received power level P_R and transmitted power level P_{MAX} as

$$P_{DESIRED} = \frac{P_{MAX}}{P_R} \times P_{MIN}$$
 where P_{MIN} is the minimum necessary received signal strength and c is a constant.

The main drawback in this protocol basic scheme is may possibility of collision, that will be observed in figure 3.28.

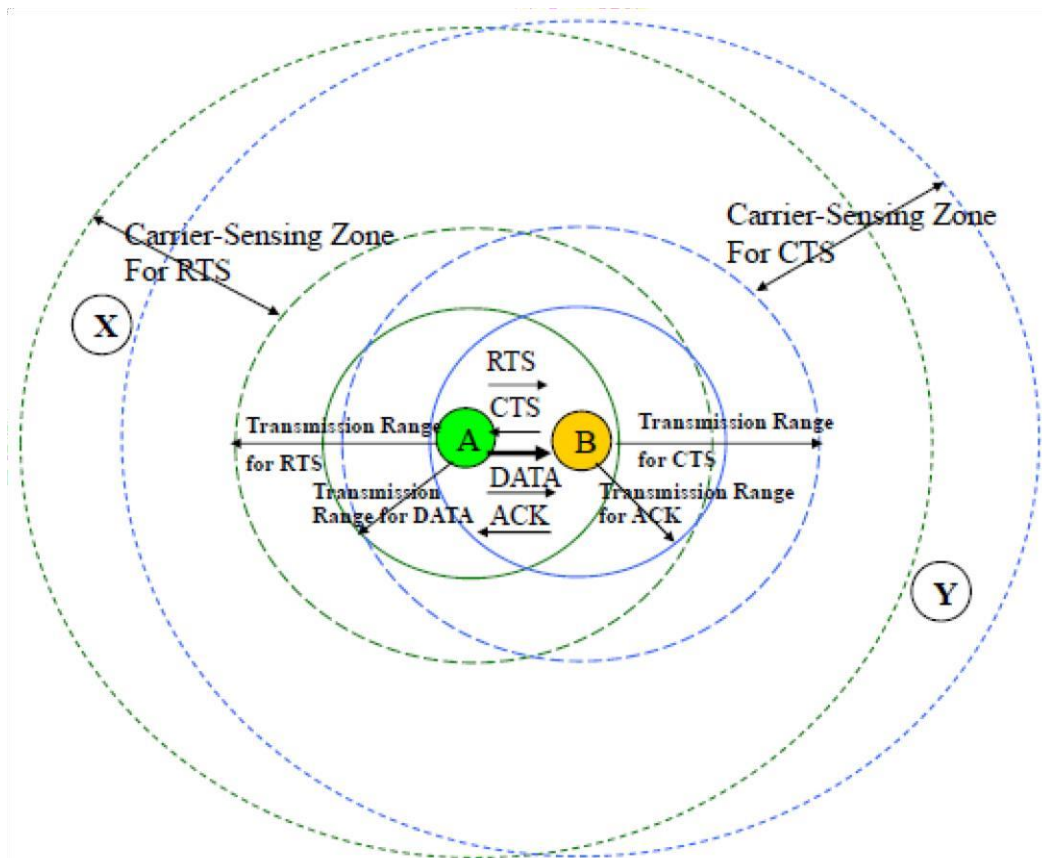


Figure 3.28 Packet transmission in BASIC scheme.

PCM modifies this scheme so as to minimize the probability of collisions. The source and receiver nodes transmit the RTS and CTS packets, as usual, with maximum power P_{MAX} . Nodes in the carrier-sensing zones of the source and receiver nodes set their NAVs (Network Allocation Vector) for EIFS (Extended Inter-Frame Space) duration when they sense the signal but are not able to decode it. In order to avoid collisions with packets transmitted by the nodes in its carrier-sensing zone, the source node transmits the DATA packet at maximum power level P_{MAX} periodically. The power level changes for RTS-CTS-DATA-ACK transmissions are shown in Figure 3.29. Hence with the above simple modification, the PCM protocol overcomes the problems faced in the BASIC scheme. PCM achieves throughput very close to that of the 802.11 protocol while using much less energy.

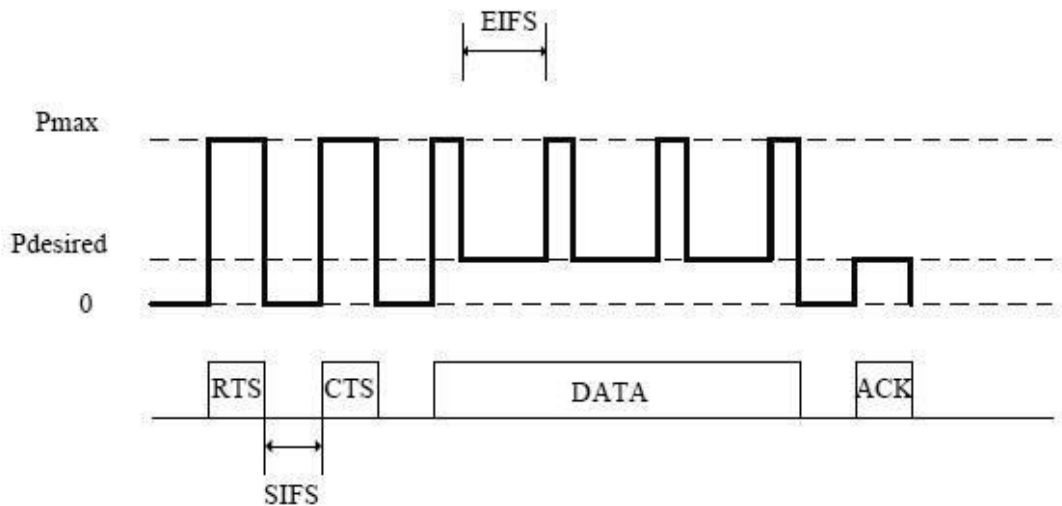


Figure 3.29 Packet transmission in PCM

3.9.4 Receiver-Based AutoRate Protocol:

The Receiver-Based AutoRate Protocol (RBAR) uses a novel rate adaptation approach. The rate adaptation mechanism is at the receiver node instead of being located at the sender. Rate adaptation is the process of dynamically switching data rates in order to match the channel conditions so that optimum throughput for the given channel conditions is achieved. Rate adaptation consists of two processes, namely, channel quality estimation and rate selection. The accuracy of the channel quality estimates significantly influences the effectiveness of the rate adaptation process. Therefore, it is important that the best available channel quality estimates are used for rate selection.

Rate selection is done at the receiver on a per-packet basis during the RTS-CTS packet exchange. Since rate selection is done during the RTS-CTS exchange, the channel quality estimates are very close to the actual transmission times of the data packets. This improves the effectiveness of the rate selection process. The RTS and CTS packets carry the chosen modulation rate and the size of the data packet, instead of carrying the duration of the reservation.

The sender node chooses a data rate based on some heuristic and inserts the chosen data rate and the size of the data packet into the RTS. When a neighbor node receives this RTS, it calculates the duration of the reservation D_{RTS} using the data rate and packet size carried on the RTS. The neighbor node then updates its NAV accordingly to reflect the reservation. Neighbor nodes receiving the CTS calculate the expected duration of the transmission and update their NAVs accordingly. The source node, on receiving the CTS packet, responds by transmitting the data packet at the rate chosen by the receiver node.

If the rates chosen by the sender and receiver are different, then the reservation Duration D_{RTS} calculated by the neighbor nodes of the sender would not be valid. D_{RTS} time period, which is calculated based on the information carried initially by the RTS packet, is referred to as *Tentative Reservation*. In order to overcome this problem, the sender node sends the data packet with a special MAC header containing a *Reservation Sub Header (RSH)*.

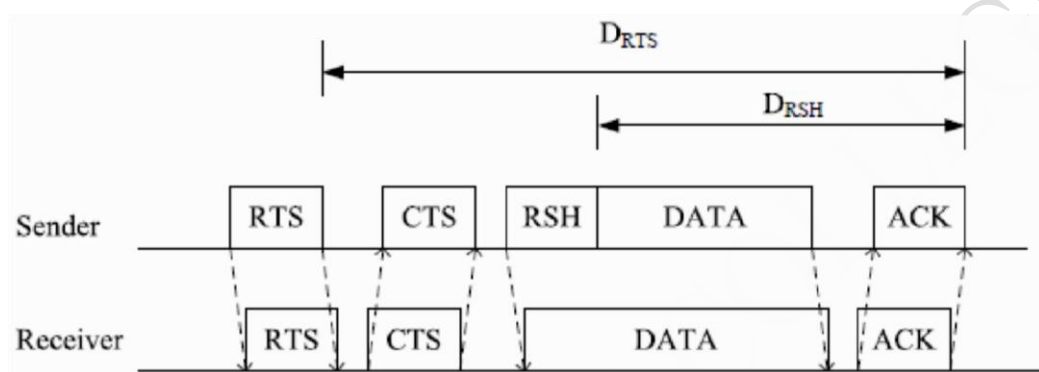


Figure 3.30 Packet transmission in RBAR

3.9.5 Interleaved Carrier-Sense Multiple Access Protocol(ICSMA): The interleaved carrier-sense multiple access protocol (ICSMA) efficiently overcomes the exposed terminal problem faced in ad hoc wireless networks. The inability of a source node to transmit, even though its transmission may not affect other ongoing transmissions, is referred to as the exposed terminal problem. For example, consider the topology shown in Figure 3.31. Here, when a transmission is going from node A to node B, nodes C and F would not be permitted to transmit to nodes D and E, respectively. Node C is called a sender-exposed node, and node E is called a receiver-exposed node. The exposed terminal problem reduces the bandwidth efficiency of the system.

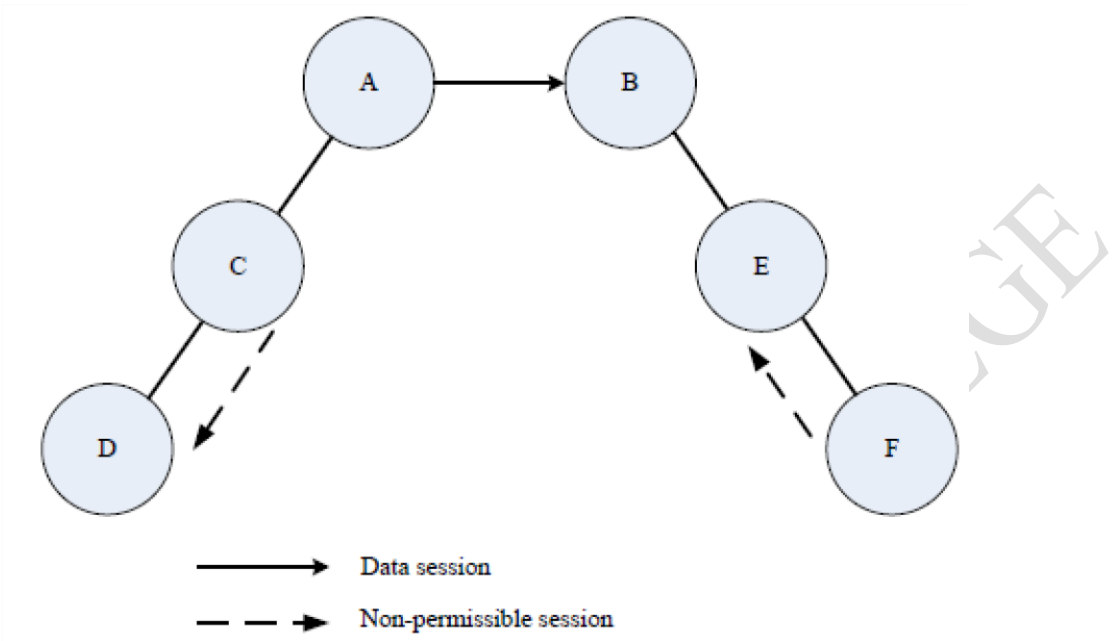


Figure 3.31 Exposed terminal problem

Each node maintains a structure called extended network allocation vector (E-NAV)

1. Two linked lists of blocks (SEList and REList):
2. List looks like $s_1, f_1; s_2, f_2; \dots; s_k, f_k$, where s_i denotes start time of the i -th block list and f_i denotes finish time of the i -th block.
3. SEList : the node would be sender-exposed in the future such that $s_j < t < f_j$
4. REList : the node would be receiver-exposed in the future such that $s_j < t < f_j$
5. Both lists are updated when RTS and CTS packets are received by the node

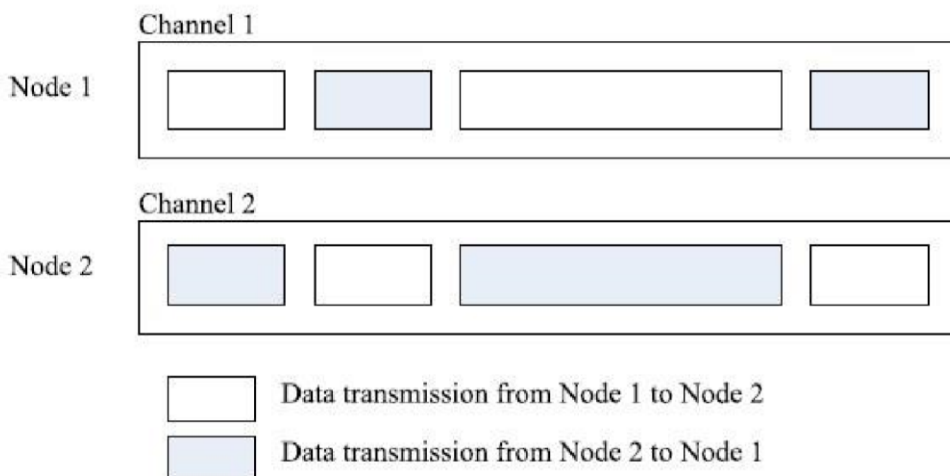


Figure 3.32 Packet transmission in ICSMA

ICSMA is a simple two-channel MAC protocol for ad hoc wireless networks that reduces the number of exposed terminals and tries to maximize the number of simultaneous sessions.

Unit IV

4. Infrastructure Establishment:

When a sensor network is first activated, various tasks must be performed to establish the necessary infrastructure that will allow useful collaborative work to be performed.

- Each node must discover which other nodes it can directly communicate with, and its radio power must be set appropriately to ensure adequate connectivity.
- Nodes near one another may wish to organize them-selves into clusters, so that sensing redundancy can be avoided and scarce resources, such as radio frequency, may be reused across non-overlapping clusters.

The common techniques used to establish such infrastructure: Topology control, Clustering, Time synchronization, and Localization for the network nodes

4.1 Topology Control:

A sensor network node that first wakes up executes a protocol to discover which other nodes it can communicate with (bidirectionally). This set of neighbors is determined by the radio power of the nodes as well as the local topography and other conditions that may degrade radio links. Unlike wired networks, nodes in a wireless sensor network can change the topology of the network by choosing to broadcast at less than their maximum possible power. This can be advantageous in situations where there is dense node deployment, as radio power is the main drain on a node's batteries. The problem of *topology control* for a sensor network is how to set the radio range for each node so as to minimize energy usage, while still ensuring that the communication graph of the nodes remains connected and satisfies other desirable communication properties.

Although in principle the transmitting range of each node can be set independently, let us first examine the simpler case where all nodes must use the same transmission range: inexpensive radio transmitters, for example, may not allow the range to be adjusted. We also ignore all effects of interference or multipath, so that any pair of nodes within range of each other can communicate. This *homogeneous* topology control setting defines the *critical transmitting range (CTR)* problem: compute the minimum common transmitting range r such that the network is connected.

The solution to the CTR problem depends on information about the physical placement of the nodes—another of the infrastructure establishment tasks. If the node locations are known a priori, or determined using the techniques described later in this chapter (see Section 4.4), then the CTR problem has a simple answer: the critical transmitting range is the length of the longest edge of the minimum Euclidean spanning tree (MST) connecting the nodes. This easily follows from the property that the MST contains the shortest edge across any partition of the nodes. The MST can be computed in a distributed fashion, using one of several such algorithms in the literature.

The CTR problem has also been studied in a probabilistic context, where the node positions are not known but their locations come from a known distribution. The problem now becomes to estimate the range r that guarantees network connectivity with high probability (probability that tends to 1 as the number n of nodes grows to infinity).

Such results are useful in settings where the node capabilities and mode of deployment prevent accurate localization. The probabilistic theory best suited to the analysis of CTR is the theory of *geometric random graphs* (GRG). In the GRG setting, n points are distributed into a region according to some distribution, and then some aspect of the node placement is investigated. If n points are randomly and uniformly distributed in the unit square, then the critical transmission range is, with high probability, $r = c\sqrt{\log n}$

n

for some constant $c > 0$. Such asymptotic results can help a node designer set the transmission range in advance, so that after deployment the network will be connected with high probability.

Most situations, however, can benefit from allowing nodes different transmission ranges. Intuitively speaking, one should choose short ranges in areas of high node density and long ranges in regions of low density. If nodes can have different transmission ranges, then the goal becomes to minimize $\sum_{i=1}^n r_i^\alpha$

1

where r_i denotes the range assigned to node i and α is the exponent describing the power consumption law for the system. This is the *range assignment* problem. A factor 2 approximation can be computed by first building an MST on the nodes, where the weight of the edge connecting nodes i and j is $\delta^\alpha(i, j)$ [here $\delta(i, j)$ denotes the Euclidean distance from i to j]. The range r_i for node i is then set to be the maximum of $\delta(i, j)$ over all nodes j which are neighbors of i in the MST [116].

The homogeneous or non-homogeneous MST-based algorithms can be expensive to implement on typical sensor nodes. Several protocols have been proposed that attempt to directly solve the CTR problem in a distributed way. For example, the COMPOW protocol of [166] computes routing tables for each node at different power levels;

- A node selects the minimum transmit power so that its routing table contains all other nodes.
- Recent work has also focused on topology control protocols that are lightweight and can work with weaker information than full knowledge of the node positions.
- For an excellent survey of these protocols and the entire topology control area, the reader is referred to [201].

4.2. Clustering:

The nodes in a sensor network often need to organize themselves into clusters. Clustering allows hierarchical structures to be built on the nodes and enables more efficient use of scarce resources, such as frequency spectrum, bandwidth, and power. It allows the health of the network to be monitored and misbehaving nodes to be identified.

The networks can be comprised of mixtures of nodes, including some that are more powerful or have special capabilities, such as increased communication range, GPS, and the like. These more capable nodes can naturally play the role of *cluster-heads*.

Each node nominates as a cluster-head the highest ID node it can communicate with (including itself). Nominated nodes then form clusters with their nominators.

Nodes that can communicate with two or more cluster-heads may become *gateways*—nodes that aid in passing traffic from one cluster to another. In some applications, it may be useful to view the IDs as weights, indicating which nodes are to be favoured in becoming cluster-heads.

- Clustering can be used to thin out parts of the network where an excessive number of nodes may be present.
- A simplified long-range communication network can be set up using only cluster-heads and gateways all other nodes communicate via their clusterhead.
- Cluster-heads can be chosen to have a minimum separation comparable to the node communication range.
- This property ensures that each cluster-head has a bounded number of cluster-head neighbors and that the density of cluster-heads is bounded from neighbors,.

Additional research is needed into how to get all the benefits of clustering while distributing the load (and battery drain) of being a cluster-head evenly among all the nodes.

4.3 Time Synchronization

Since the nodes in a sensor network operate independently, their clocks may not be, or stay, synchronized with one another. This can cause difficulties when trying to integrate and interpret information sensed at different nodes. For example, if a moving car

is detected at two different times along a road, before we can even tell in what direction the car is moving, we have to be able to meaningfully compare the detection times.

For instance, many localization algorithms use ranging technologies to estimate internodes distances; in these technologies, synchronization is needed for time-of-flight measurements that are then transformed into distances by multiplying with the medium propagation speed for the type of signal used (say, radio frequency or ultrasonic). Configuring a beam-forming array or setting a TDMA radio schedule are just two more examples of situations in which collaborative sensing requires the nodes involved to agree on a common time frame.

While in the wired world time synchronization protocols such as NTP [159, 160] have been widely and successfully used to achieve *Coordinated Universal Time* (UTC), these solutions do not transfer easily to the ad hoc wireless network setting.

These wired protocols assume the existence of highly accurate master clocks on some network nodes (such as atomic clocks) and, more importantly, they also require that pairs of nodes in the protocol are constantly connected and that they experience consistent communication delays in their exchanges.

Unfortunately, none of these assumptions is generally valid in sensor networks. No special master clocks are available, connections are ephemeral, and communication delays are inconsistent and unpredictable. Thus we must moderate our goals when it comes to synchronizing node clocks in sensor networks.

4.3.1 Clocks and Communication Delays

Computer clocks are based on hardware oscillators which provide a local time for each sensor network node. At real time t the computer clock indicates time $C(t)$, which may or may not be the same as t . For a perfect hardware clock, the derivative $dC(t)/dt$ should be equal to 1. If this is not the case, we speak of clock *skew* (also called *drift*). The clock skew can actually change over time due to environmental conditions, such as temperature and humidity, but we will assume it stays bound close to 1, so that

$$1 - \rho \leq \frac{dC(t)}{dt} \leq 1 + \rho$$

Where ρ denotes the maximum skew. A typical value of ρ for today's hardware is 10^{-6} . Small fluctuations on the skew are usually modeled as random Gaussian noise. Note that, because of clock skew, even if the clocks of two nodes are synchronized at some point in time, they need not stay so in the future.

Even if no skew is present, the clocks of different nodes may disagree on what time

“0” means. Time differences caused by the lack of a common time origin are referred to as clock *phase* differences (or clock *bias*).

Send time: This is the time taken by the sender to construct the message, including delays introduced by operating system calls, context switching, and data access to the network interface.

Access time: This is the delay incurred while waiting for access to the transmission channel due to contention, collisions, and the like. The details of that are very MACspecific.

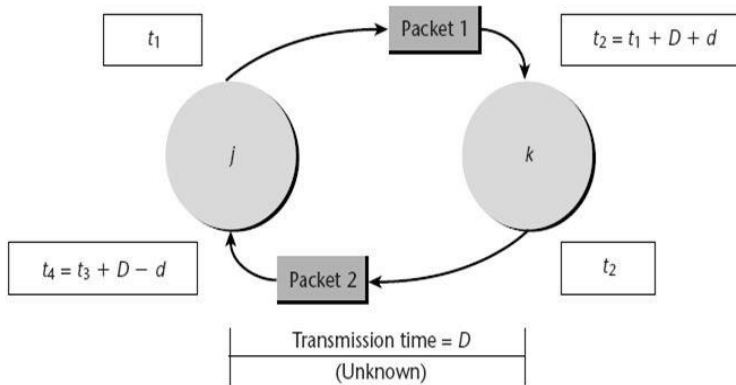
Propagation time: This is the time for the message to travel across the channel to the destination node. It can be highly variable, from negligible for single-hop wireless transmission to very long in multihop wide-area transmissions.

Receive time: This is the time for the network interface on the receiver side to get the message and notify the host of its arrival. This delay can be kept small by time-stamping the incoming packet inside the network driver’s interrupt handler.

Node i reads its local clock with time value t_1 and sends this in a packet to node j .

Node j records the time t_2 according to its own clock when the packet was received. We must have $t_2 = t_1 + D + d$. Node j , at time t_3 , sends a packet back to i containing t_1 , t_2 , and t_3 .

Node j , at time t_3 , sends a packet back to i containing t_1 , t_2 , and t_3 .



Clock phase difference estimation, using three message exchanges

Figure 4.1 Clock phase difference estimation, using three message exchanges (adapted from [104]).

Node i receives this packet at time t_4 . We must have $t_4 = t_3 + D - d$. Therefore, node i can eliminate D from the above two equations and compute $d = (t_2 - t_1 - t_4 + t_3)/2$.

Finally, node i sends the computed phase difference d back to node j .

Time synchronization can then be propagated across the network by using a spanning tree favouring direct connections with reliable delays [220]. In the presence of clock skew, however, frequent resynchronizations may be required. Furthermore, such ideal conditions on delays are hardly ever true in a sensor network.

4.3.2 Interval Methods:

As we mentioned, in many situations involving temporal reasoning, the temporal ordering of events matters much more than the exact times when events occurred. In such situations, interval methods provide a lightweight protocol that can be used to move clock readings around the network and perform temporal comparisons [194].

Suppose that event E occurs at real time $t(E)$ and is sensed by some node i and given a time stamp $S_i(t)$, according to the local clock of node i . Suppose also that node i 's out—

but clock skew and network latency have to be dealt with. We will call intervals between events *durations*.

In the simplest setting, if node 1 with maximum clock skew ρ_1 wishes to transform a local duration C_1 into the time framework of node 2 with maximum clock skew ρ_2 , we can proceed as follows. If the real time duration is t , then we must have

$$1 - \rho_i \leq \frac{\Delta C_i}{\Delta t} \leq 1 + \rho_i,$$

for $i = 1, 2$. Thus the real time duration t is contained in the interval $[C_1/(1 + \rho_1), C_1/(1 - \rho_1)]$, and the duration according to the clock of node 2 satisfies

Now suppose nodes 1 and 2 are neighbors and have a direct communication link between them. Node 1 has detected an event E and time-stamped it with time stamp $r_1 =$

$S_1(E)$. This temporal event needs to be communicated and transformed into the temporal frame of node 2. We must estimate the communication delay between the nodes. Now, under most communication protocols, for every message M that node 1 sends to node 2, there is a return acknowledgment message A from node 2 to node 1. Node 1 can measure the duration d between transmitting M and receiving A and use that as an upper bound on the communication delay (the obvious lower bound is 0). However, it is node 2 that needs to know this information in order to update the time stamp generated by node 1. This seems to require two message exchanges between nodes 1 and 2: a message M_1 carrying r_1 , and a subsequent message M_2 carrying d (along with the corresponding acknowledgments A_1 and A_2), thus effectively doubling the communication overhead (see Figure 4.2).

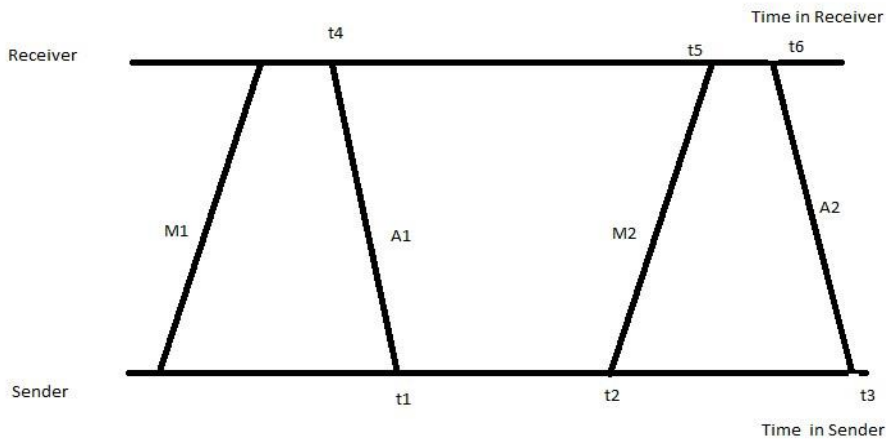


Figure 4.2 Transforming time stamps from the reference frame of one node to that of another (adapted from [194]).

Because communicating nodes typically exchange several messages, it makes sense to piggyback delay information on other content-carrying messages. Let us look again at Figure 4.2. Suppose $M1$ was a message sent from node 1 to node 2 earlier, for other purposes. Now $M2$ will be used to carry information about the time stamp $S1(E)$. The idle time duration $1 = t2 - t1$ can be measured according to the local clock of node 1, as the time between receiving $A1$ and transmitting $M2$. The round-trip duration $p1 = t5 - t4$ can be measured according to the local clock of node 2, as the time between transmitting $A1$ and receiving $M2$. If sender node 1 piggybacks the idle time duration 1 on $M2$, then at time $t5$, the receiver node 2 can estimate the communication delay d via the bounds

$$0 \leq d \leq p_1 - l_1$$

$$1 - p_2$$

$$1 + p_1$$

The two nodes had earlier communication in the recent past and that, if a node communicates with several others, it keeps track of its last communication to each of its neighbors. Time stamps can be propagated from node to node as follows. Let r_i and s_i denote, respectively, the times when node i receives and sends out the packet containing the time stamp (measured according to its local clock). Let l_i and p_i denote the corresponding idle and round-trip times, as earlier (note that l_i is measured in the clock of

node $i - 1$). Then we can recursively maintain a valid interval guaranteed to contain the original time stamp according to the local clock of node i , as follows.

For node 1, the interval is $[r_1, r_1] = [S_1(E), S_1(E)]$.

For node 2, using the above reasoning, the interval is

And for the n^{th} node in the transmission chain we get by iterating

Thus in the end, detection times at one node can be transformed to time intervals in the local time frame of another node through a sequence of one-hop communications. Comparison of time stamps is done through standard interval methods: if the corresponding intervals are disjoint, then a meaningful time-stamp comparison can be made, and otherwise not.

This time synchronization protocol has low overhead, scales well with network size, and can accommodate topology changes and short-lived connections. But, as with most interval methods, the intervals computed can get too large to be useful or to resolve most of the time comparisons needed.

4.3.3 Reference Broadcasts

Time comparisons are not sufficient for all applications and map-pings from event times to time intervals may quickly become useless if long delays or multihop routes increase the interval sizes beyond reasonable limits. Note that even if there is no skew between the clocks of different nodes (say $\rho_i = 1$ for all nodes above), time intervals can still grow large because delay estimates can have large uncertainty.

The key idea of the *reference broadcast system* (RBS) of [63] is to use the broadcast nature of the wireless communication medium to reduce delays and delay uncertainty in the synchronization protocol. This is achieved by having the receiver nodes within the communication range of a broadcast message sent by a sender node synchronize with one another, rather than with the sender. This is accomplished by having the sender send a reference message to receivers who record its time of arrival each in their own time frame. The receivers then exchange this information among themselves. Receive times can be subject to random fluctuations because of environmental conditions; one way to mitigate the impact of these nondeterministic variations is to repeat the reference broadcast protocol a number of times and then average results in computing internode time offsets (relative phase differences). The group dispersion (i.e., the maximum offset between any pair of receiver nodes) can be significantly decreased in a large group of receiver nodes by such statistical methods (say, by a factor of 5, by repeating the reference broadcast 20 times [63]).

The RBS protocol as described up until now deals with synchronization only among nodes that are in range of the same sender. Multihop synchronization is also possible, by just composing the inter-receiver clock affine mappings in the right way. Consider the scenario in Figure 4.3.

In this figure, sender *A* can synchronize nodes 1, 2, 3, and 4, while sender *B* can synchronize nodes 4, 5, 6, and 7 (of course, the sender/receiver distinction is purely artificial—any node can be sender or receiver). The affine clock maps obtained by RBS between nodes 1 and 4 (through *A*), and nodes 4 and 7 (through *B*), for example, can be composed to provide the clock map between nodes 1 and 7. In general, we can imagine an RBS graph with an edge

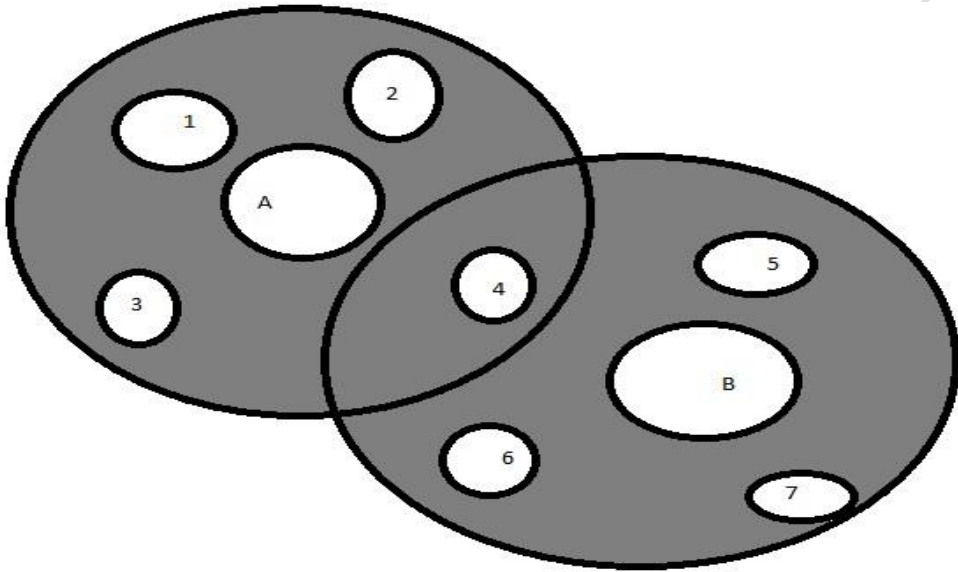


Figure 4.3 The restricted broadcast time synchronization protocol between non-neighboring nodes (adapted from [63]).

Between any two nodes that can be reached in one hop from a common sender. The edge between these nodes can be given a weight corresponding to the uncertainty of the clock phase difference estimation between the two nodes. Then, whenever two nodes need to synchronize in the graph, a shortest path between them can be sought using standard shortest-path graph algorithms [46]. Such algorithms can be expensive to run in a distributed sensor network setting; an alternative is to use the RBS pair wise affine clock map-pings for neighbouring nodes to transform times in a packet to the local frame of the current receiver, as the packet travels through the network, much in the spirit of the interval methods described in Section 4.3.2.

4.4 Localization and Localization Services

An attractive feature of a sensor network is that it can provide information about the world that is highly localized in space and/or time. For several sensor net applications, including target tracking and habitat monitoring, knowing the exact location where information was collected is critical. In fact, for almost all sensor net applications, the value of the information collected can be enhanced if the location of the sensors where readings were made is also available. With current technologies, most sensor nodes remain static. Thus one way to know the node positions is to have the network installer measure these

locations during network deployment. This may not always be feasible, however, especially in ad hoc deployment scenarios (such as dropping sensors from an aircraft), or in situations where the nodes may need to be moved for a variety of reasons (or move themselves, when that capability can be incorporated).

In this section, we describe techniques for *self-localization*—that is, methods that allow the nodes in a network to determine their geographic positions on their own as much as possible, during the network initialization process. We also describe *location service* algorithms—methods that allow other nodes to obtain the location of a desired node, after the initial phase in which each node discovers its own location. Such location services are important for geographic routing, location-aware query processing, and many other tasks in a sensor network.

Since the availability of GPS systems in 1993, it has been possible to build autonomous nodes that can localize themselves within a few meters' accuracy by listening to signals emitted by a number of satellites and assistive terrestrial transmitters. But even today GPS receivers can be expensive and difficult to incorporate into every sensor node for a number of practical reasons, including cost, power consumption, large form factors, and the like. Furthermore, GPS systems do not work indoors, or under dense foliage, or in other expectable conditions. Thus in a sensor network context, it is usually reasonable to assume that some nodes are equipped with GPS receivers, but most are not. The nodes that know their position are called *landmarks*. Other nodes localize themselves by reference to a certain number of landmarks, using various ranging or direction-of-arrival technologies that allow them to determine the distance or direction of landmarks.

4.4.1 Ranging Techniques

Ranging methods aim at estimating the distance of a receiver to a transmitter, by exploiting known signal propagation characteristics. For example, pairs of nodes in a sensor network whose radios are in communication range of each other can use *received signal strength* (RSS) techniques to estimate the RF signal strength at the receiver.

A second way to estimate distance is to measure the time it takes for a signal to travel from sender to receiver; this can be multiplied by the signal propagation speed to yield distance. Such methods are called *time of arrival* (TOA) techniques and can use either RF or ultrasound signals. This requires that the sender and receiver are synchronized and that the sender knows the exact time of transmission and sends that to the receiver.

An alternative is to measure the *time difference of arrival* (TDOA) at two receivers, which then lets us estimate the difference in distances between the two receivers and the sender. Another issue is that signal propagation speed exhibits variability as a function of temperature or humidity as well (especially for ultrasound), and thus it is not realistic to assume it is constant across a large sensor field. Local pairs of beacons can be used to estimate local propagation speed. With proper calibration and the best current techniques, localization to within a few centimetres can be achieved.

4.4.2 Range-Based Localization Algorithms

We now describe methods for localizing sensor network nodes with reference to nearby *landmarks*—we shall use the latter term to refer to other nodes that have already been localized. We confine our attention to distance measurements, obtained using one of the ranging techniques described earlier.

The position of a node in the plane is determined by two parameters: its x and y coordinates. Therefore, at least two constraints are necessary to localize a node. A distance measurement with respect to a landmark places the node on a circle centered at the landmark whose radius is the measured distance (in the TDOA case, a difference of distances to two landmarks places the node on a hyperbola with the landmarks as foci). Since quadratic curves in the plane can have multiple intersections, in general a third distance measurement is necessary in order to completely localize a node (see Figure 4.4).

In fact, most measurements have error, so the node in question is only localized to within a band around the measured distance circle. For this reason, it may be advantageous to use redundant measurements and least-squares techniques to improve the estimation accuracy. With additional measurements, in TOA methods, the propagation speed can be estimated locally as well, which will yield improved localization accuracy, as mentioned earlier.

We now describe this operation, called *atomic multilateration*, in some detail [204]. The analysis is similar to the collaborative source localization discussed in Section 2.2.2. Suppose we number the node whose location we seek as node 0 and the available landmark nodes as $1, 2, \dots, n$. Let the position of node i be (x_i, y_i) and its measured

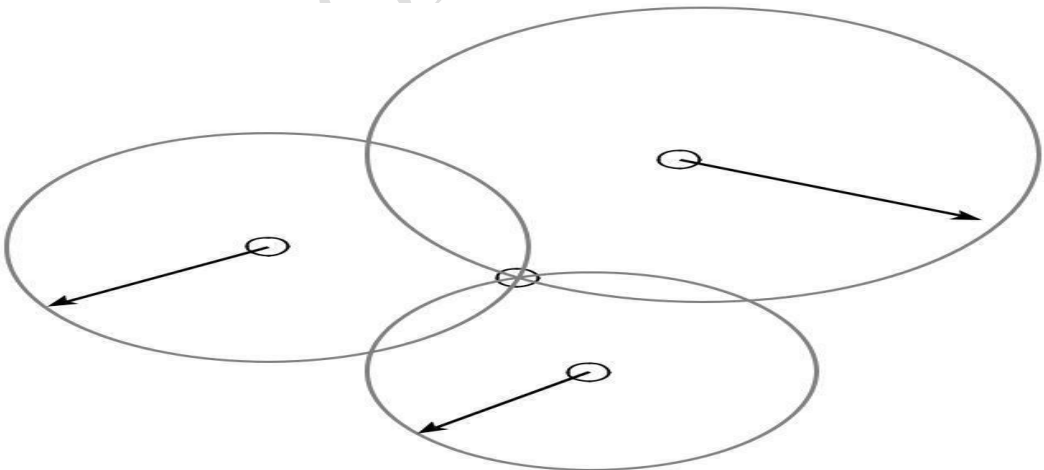


Figure 4.4 Localizing a node using three distance measurements.

time of arrival at node 0 be t_i (for $1 \leq i \leq n$). If s denotes the local signal propagation speed, then we have for each i , $1 \leq i \leq n$

$$\sqrt{(x_i - x_0)^2 + (y_i - y_0)^2} + \varepsilon_i(x_0, y_0, s) = st_i,$$

where ε_i indicates the error in the i^{th} measurement due to noise and other factors. We can also give each measurement a relative weight α_i , indicating how much confidence we want to place in it. Our goal then is to estimate x_0, y_0 , and s so as to minimize the weighted total squared error

$$E(x_0, y_0, s) = \sum \alpha_i^2 \varepsilon_i^2(x_0, y_0, s)$$

Simplicity, we assume below that we have set $\alpha_i = 1$ for all i .

We can linearize the above system of n constraints by squaring and subtracting the equation for measurement 1 from that of the others, thus obtaining $n - 1$ linear equations of the form (the $x_0^2 + y_0^2$ terms cancel):

$$2x_0(x_i - x_1) + 2y_0(y_i - y_1) + s^2(t_i^2 - t_1^2) = -x_i^2 - y_i^2 + x_1^2 + y_1^2.$$

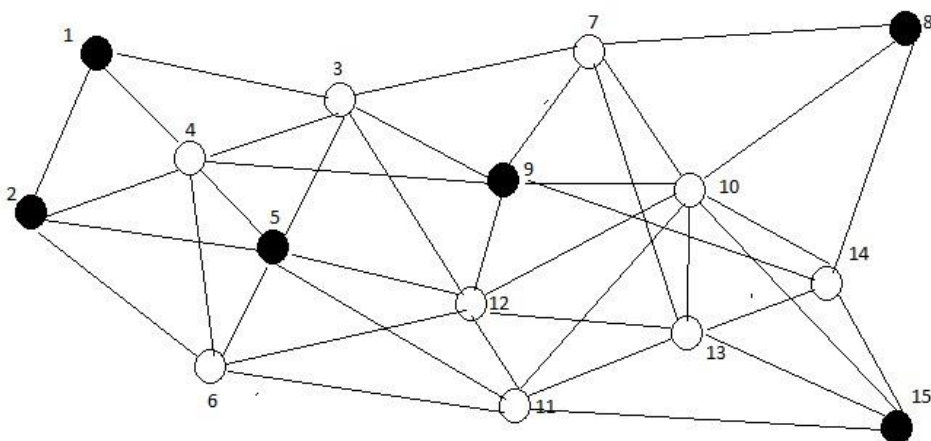


Figure 4.5 Iterative multilateration successively localizes more and more nodes (adapted from [203]).

Algorithm is executed centrally or in a distributed fashion, there will be node configurations where this process will be unable to localize all the nodes.

In the general case, iterative multilateration fails. What we have to go by is the positions of the original landmark nodes with GPS, plus various distance estimates between node pairs. In principle, if we have enough constraints and no degeneracies are present, we can write this as a large system of nonlinear equations that can be solved for the unknown node positions. However, solving such global algebraic systems is expensive, and the solution must be computed in a centralized fashion. To get around these difficulties, a method titled *collaborative multilateration* is described in [203] which admit of a reasonable distributed implementation.

4.4.3 Other Localization Algorithms

In settings where RSS and other ranging technologies cannot be used directly to estimate distances, there are a number of alternatives. In every sensor network, each node knows what other nodes it can talk to directly its one-hop neighbors. If the sensor nodes are densely and uniformly deployed, then we can use hop counts to landmarks as a substitute to physical distance estimates. In this setting, each landmark floods the network with a broadcast message whose hop count is incremented as it is passed from node to node. The hop count in the message from a landmark that first reaches a node is the hop distance of that node to the landmark (standard graph-based breadth-first search). In order to transform hop counts into approximate distances, the system must estimate the average distance corresponding to a hop. This can be done either by using inter-landmark distances that are known in both hop and Euclidean terms [169], or by using prior information about the size of the area where the nodes are deployed and their density [165]. Once a node has approximate distances to at least three landmarks this way, then the previous unit and multilateration techniques can be used.

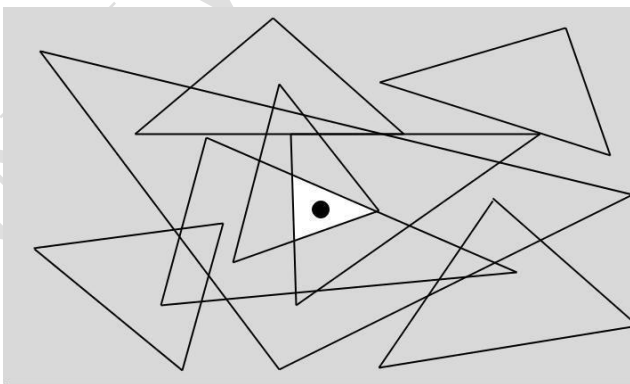


Figure 4.6 Node localization using multiple triangle containment tests (adapted from [93]).

The same paper argues that the accuracy of localization provided by k landmarks is $O(1/k^2)$.

4.4.4 Location Services

For geographic forwarding, energy-aware routing, and many other sensor net tasks, it is important to have a *location service*, a mechanism for mapping from node IDs of some sort to node locations. Note that even if the actual nodes in a sensor network are static, virtual node IDs, such as the tracking cluster leader mentioned earlier, can be mobile because of hand-offs from node to node (as the target moves).

Providing a location service is an interesting problem, because the obvious solutions of having a central repository that correlates IDs to locations, or having a copy of that at every node, suffer from significant drawbacks. The former has a single point of failure and can cause communication delays unrelated to the distance between the sender and receiver; the latter requires space on every node and is extremely expensive to update in case some (virtual or real) nodes move, or are inserted or deleted.

What we would like to have is a distributed location service that is robust to single node failures, spreads the load evenly across the network, and also has nice locality properties, in the sense that a node wishing to determine the location of another nearby node can do so at a cost that is sensitive to the distance of the receiver. Such a location service is in the *Grid* system presented in [134], which we describe below. We present this service using node UIDs as the node “names”; readers should keep in mind that any virtual name can be used equally well.

Figure 4.7 The location servers for node B , as selected by GLS (adapted from [134])

At some point, the ascending tree path from A will meet the ascending path from B that established the triplets of B 's location servers at each level. At that time, the search terminates and the location of B can be determined.

Figure 4.8 The nodes for which a given node acts as a location server are shown in small type size in the quad-tree cell of that node. Those having B 's location are shown in bold. The paths of two possible location queries for B are also shown, originating at node A (adapted from [134]).

The crucial observation here is that at each step of the search, the current node, say C , is associated with a tile at a certain level i of the quad-tree and forwards the message to a node D which is the best node in the parent tile of the tile containing C , at level $i + 1$. This is because node D must be known to C . Indeed, node D cannot lie in the same level i tile as C , because D is “better” than C . But D recruited location servers for itself at level i , including in the level i tile containing C . Since C is the smallest ID node greater than B in the tile, it is *a fortiori* the smallest ID node greater than D as well. Therefore, C was recruited by D , and D is known to C . Consider again the example in Figure 4.8. When node A is the one with ID 76, the query proceeds to nodes 21 and then 20, a node which is a location server for B . When node A is the one with ID 90, the query proceeds to nodes 70 and 37, where it terminates because again the latter is a location server for B .

This algorithm has many nice properties. In particular, if the source and destination lie in a common quad-tree tile at level i , then at most i steps are needed before a location server for the destination can be found for the source. This makes the cost of the location service distance-sensitive: the look-up time is sensitive to the separation between the source and destination. This is a reflection of the fact that a node selects more location servers near itself, and fewer and fewer as we move farther away. Thus in an area of the network where location servers for a node B are far from each other and where we may have to take many steps to find one, that is OK because node B itself is far away and the cost of reaching the server can be amortized over the cost of reaching the destination B .

Sensor Tasking and Control

To efficiently and optimally utilize scarce resources in a sensor network, such as limited on-board battery power supply and limited communication bandwidth, nodes in a sensor network must be carefully tasked and controlled to carry out the required set of

tasks while consuming only a modest amount of resources. For example, a camera sensor may be tasked to look for animals of a particular size and color, or an acoustic sensor may be tasked to detect the presence of a particular type of vehicle. To detect and track a moving vehicle, a pan-and-tilt camera may be tasked to anticipate and follow the vehicle object. It should be noted that to achieve scalability and autonomy, sensor tasking and control have to be carried out in a distributed fashion, largely using only local information available to each sensor.

For a given sensing task, as more nodes participate in the sensing of a physical phenomenon of interest and more data is collected, the total utility of the data, perhaps measured as the information content in the data, generally increases. However, doing so with all the nodes turned on may consume precious battery power that cannot be easily replenished and may reduce the effective communication bandwidth due to congestion in the wireless medium as well. Furthermore, as more nodes are added, the benefit often becomes less and less significant, as the so-called diminishing marginal returns set in, as shown in Figure 5.1. To address the balance between utility and resource costs, this chapter introduces a utility-cost-based approach to distributed sensor network management.

After discussing the general issues of task-driven sensing (Section 5.1), we develop a generic model of utility and cost (Section 5.2). Next, we present the main ideas of information-based sensor tasking

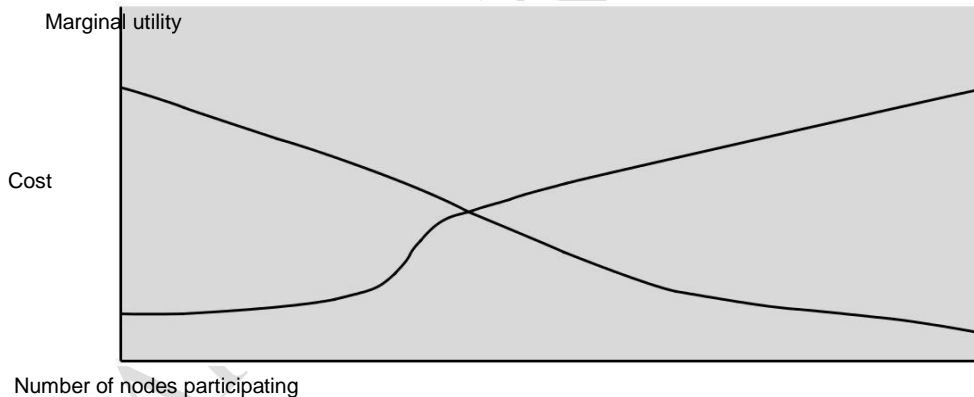


Figure 5.1 Utility and cost trade-off: As the number of participating nodes increases, we see diminishing returns.

and discuss a specific realization in information-driven sensor query-ing (IDSQ) and a cluster-leader based protocol in which information about a physical phenomenon resides at a fixed leader for an extended period of time (Section 5.3). We then introduce dynamic migration of information within a sensor network, as in the case of tracking a moving phenomenon. Here, we will address the issues of information hand-off, greedy versus multistep lookahead information routing, as well as maintenance of collaborative processing sensor groups (Section 5.4). Although the material in this chapter is introduced in the context of localization or tracking tasks, the basic idea of information-based sensor tasking is applicable to more general sensing tasks.

5.1 Task-Driven Sensing

The purpose of a sensor system is often viewed as obtaining information that is as extensive and detailed as possible about the unknown parts of the world state. Any targets present in the sensor field need to be identified, localized, and tracked. All this data is to be centrally aggregated and analyzed. This is a reasonable view when the potential use of this information is not known in advance, and when the cost of the resources needed to acquire and transmit the information is either fixed or of no concern. Such a scheme, however, runs the danger of flooding the network with useless data and depleting scarce resources from battery power to human attention, as already mentioned. There are obvious ways to be more selective in choosing what sensor nodes to activate and what information to communicate; protocols such as directed diffusion routing (see Section 3.5.1) address exactly this issue for the transport layer of the network.

When we know the relevant manifest variables defining the world state—say, the position and identity of each target—then computing the answers to queries about the world state is a standard algorithm design problem. An algorithm typically proceeds by doing both numerical and relational (e.g., test) manipulations on these data, in order to compute the desired answer. The quality of the algorithm is judged by certain performance measures on resources, such as time and space used.

5.2 Roles of Sensor Nodes and Utilities

Sensors in a network may take on different roles. Consider the following example of monitoring toxicity levels in an area around a chemical plant that generates hazardous waste during processing. A number of wireless sensors are initially deployed in the region [see Figure 5.2(a)]. Due to the nature of the environment and the cost of deployment, further human intervention or node replacement is not feasible. The sensors form a mesh network, and data collected by a subset of nodes is transmitted, through the multihop network,

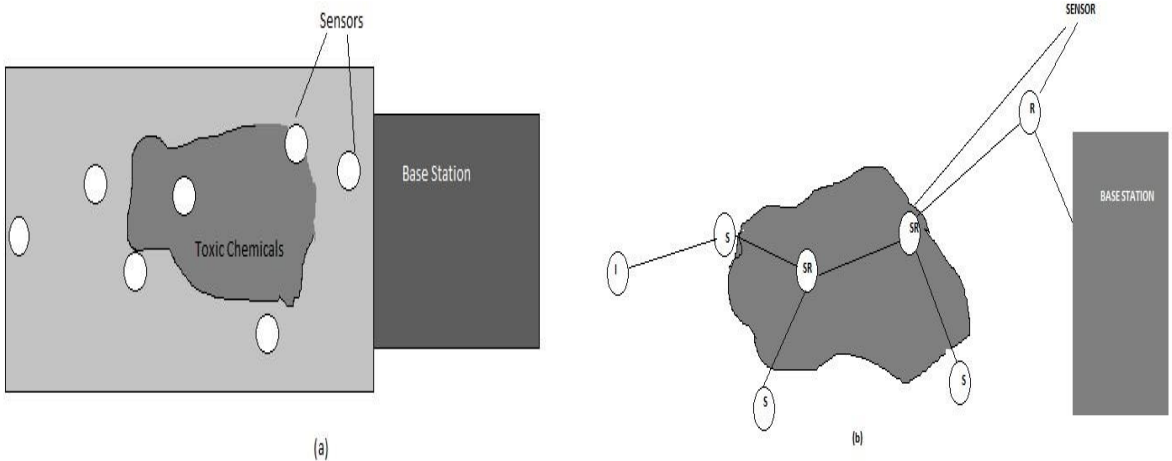


Figure 5.2 Sensor tasking: (a) A chemical toxicity monitoring scenario. (b) Sensors may take on different roles such as sensing (S), routing (R), sensing and routing (SR), or being idle (I), depending on the tasks and resources.

and relayed off the network to an adjoining base station or gateway. The network may be tasked to monitor the maximum toxicity levels in the region.

A sensor may take on a particular role depending on the application task requirement and resource availability such as node power levels [Figure 5.2(b)]. Some of the nodes, denoted by SR in the figure, may participate in both sensing and routing. Note that routing includes both receiving and transmitting data. Some (S) may perform sensing only and transmit their data to other nodes. Some (R) may decide to act only as routing nodes, especially if their energy reserve is limited. Yet still others (I) may be in an idle or sleep mode, to preserve energy. As one can see, as the node energy reserve or other conditions change, a sensor may take on a different role. For example, a sensing-and-routing sensor may decide to drop the sensing role as soon as its energy reserve is below a certain level. To study the problem of determining what role a sensor should play, we first introduce utility and cost models of sensors and then techniques that find optimal or nearly optimal assignments. A utility function assigns a scalar value, or utility, to each data reading of a sensing node; that is,

$$U: I \times T \rightarrow R$$

Where $I = \{1, \dots, K\}$ are sensor indices and T is the time domain. Each sensor operation is also assigned a cost. The cost of a sensing operation is C_s , aggregation cost is C_a , transmission cost is C_t , and reception cost is C_r . Note that these are unit costs per datum or packet, assuming the data in each operation can be so encapsulated. We further denote the set of nodes performing a sensing operation at time t as $V_s(t)$, aggregation nodes as $V_a(t)$, transmitting nodes as $V_t(t)$, and receiving nodes as $V_r(t)$. Omitting the issue of communication channel access contention and the possibility of retransmission

Determine the sets of sensors V_s , V_t , V_r , and V_a that maximize the utility over a period of time subject to the constraint

$$\begin{aligned} \text{Max } & \sum_t \sum_{i \in V_s(t)} U(i,t) \\ \sum_t & \sum_{i \in V_s(t)} C_s + \sum_t \sum_{i \in V_t(t)} (C_t + C_r) + \sum_t \sum_{i \in V_a(t)} C_a \leq C_{\text{total}} \end{aligned}$$

We make a number of observations about the structure of utility and cost models. The utility of the network depends on the underlying routing structure. In a routing-tree realization, the tree must span the nodes that appear in the utility function and the base

5.3 Information-Based Sensor Tasking

The main idea of information-based sensor tasking is to base sensor selection decisions on information content as well as constraints on resource consumption, latency, and other costs. Using information utility measures, sensors in a network can exploit the information content of data already received to optimize the utility of future sensing and communication actions, thereby efficiently managing scarce communication and processing resources. For example, IDSQ [233, 43] formulates the sensor tasking problem as a general distributed constrained optimization that maximizes information gain of sensors while minimizing communication and resource usage. We describe the main elements of the information-based approaches here.

5.3.1 Sensor Selection

Recall from Chapter 2 that for localization or tracking problem, a belief refers to the knowledge about the target state such as position and velocity. In the probabilistic framework, this belief is represented as a probability distribution over the state space. We consider two scenarios, localizing a stationary source and tracking a moving source, to illustrate the use of information-based sensor tasking.

In the first scenario, a leader node might act as a relay station to the user, in which case the belief resides at this node for an extended time interval, and all information has to travel to this leader. In the second scenario, the belief itself travels through the network, and nodes are dynamically assigned as leaders. In this section, we consider the fixed leader

protocol for the localization of a stationary source and postpone the discussion of the moving leader protocols to Section 5.4.

Given the current belief state, we wish to incrementally update the belief by incorporating the measurements of other nearby sensors. However, not all available sensors in the network provide useful information that improves the estimate. Furthermore, some information may be redundant. The task is to select an optimal subset and an optimal order of incorporating these measurements into our belief update. Note that in order to avoid prohibitive communication costs, this selection must be done without explicit knowledge of measurements residing at other sensors. The decision must be made solely based upon known characteristics of other sensors, such as their position and sensing modality, and predictions of their contributions, given the current belief about the phenomenon being monitored.

Figure 5.3 illustrates the basic idea of sensor selection. The illustration is based on the assumption that estimation uncertainty can be effectively approximated by a Gaussian distribution, illustrated by uncertainty ellipsoids in the state space. In the figure, the ellipsoid at time t indicates the residual uncertainty in the current belief state. The ellipsoid at time $t + 1$ is the incrementally updated belief after incorporating an additional sensor, either a or b , at the next time step. Although in both cases, a and b , the area of high uncertainty is reduced by the same percentage, the residual uncertainty in case a maintains the largest principal axis of the distribution. If we were to decide between the two sensors, we might favor sensor b over sensor a , based on the underlying measurement task.

Although details of the implementation depend on the network architecture, the fundamental principles derived in this chapter hold for both the selection of a remote sensor by a cluster-head as well

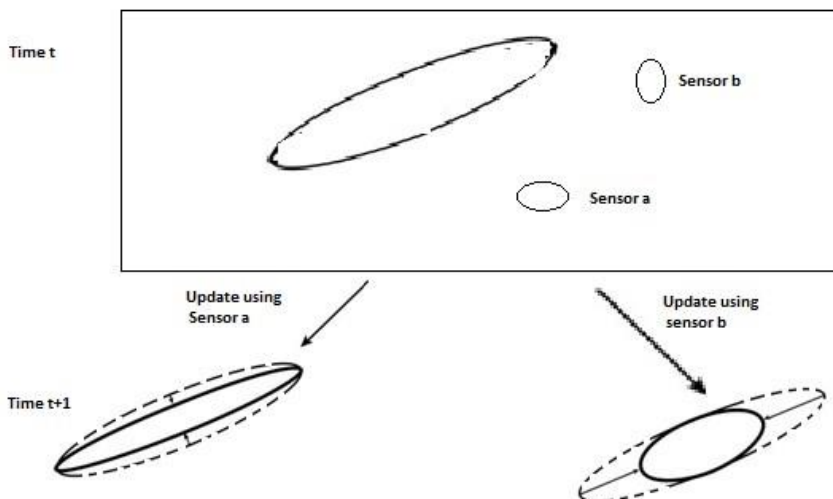


Figure 5.3 Sensor selection based on information gain of individual sensor contributions. The information gain is measured by the reduction in the error ellipsoid. In the figure, reduction along the longest axis of the error ellipsoid produces a larger improvement in reducing uncertainty. Sensor placement geometry and sensing modality can be used to compare the possible information gain from each possible sensor selection, a or b .

as the decision of an individual sensor to contribute its data and to respond to a query traveling through the network. The task is to select the sensor that provides the best information among all available sensors whose readings have not yet been incorporated. As will be shown in the experimental results, this provides a faster reduction in estimation uncertainty and usually incurs lower communication overhead for meeting a given estimation accuracy requirement, compared with blind or nearestneighbor sensor selection schemes.

Example: Localizing a Stationary Source

In this example, 14 sensors are placed in a square region, as shown in Figure 5.4. Thirteen sensors are lined along the diagonal, with one sensor off the diagonal near the upper left corner of the square. The true location of the target is denoted by a cross in the figure.

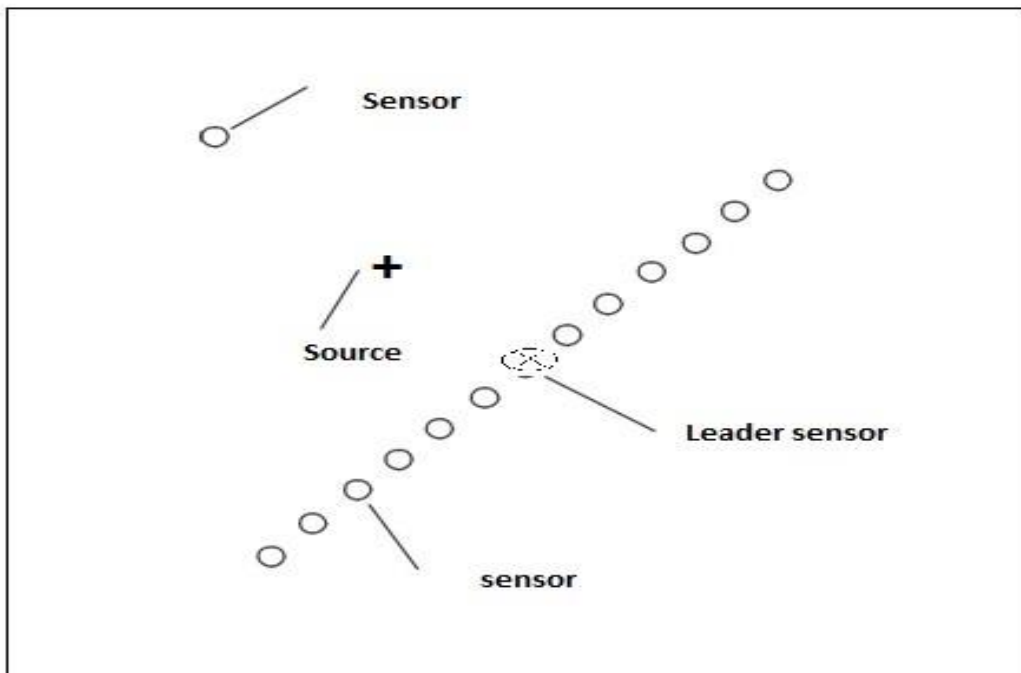


Figure 5.4 Source localization. The leader node (denoted by \times) queries other sensors (circles), to localize the source marked by a cross (+).

To illustrate sensor selection among one-hop neighbors, we assume all sensors can communicate to the node at the center, which queries all other nodes and acts as the data fusion center (the leader). We further assume that each sensor's measurement provides an estimate of the distance to the target, in the form of a doughnut-shaped likelihood function

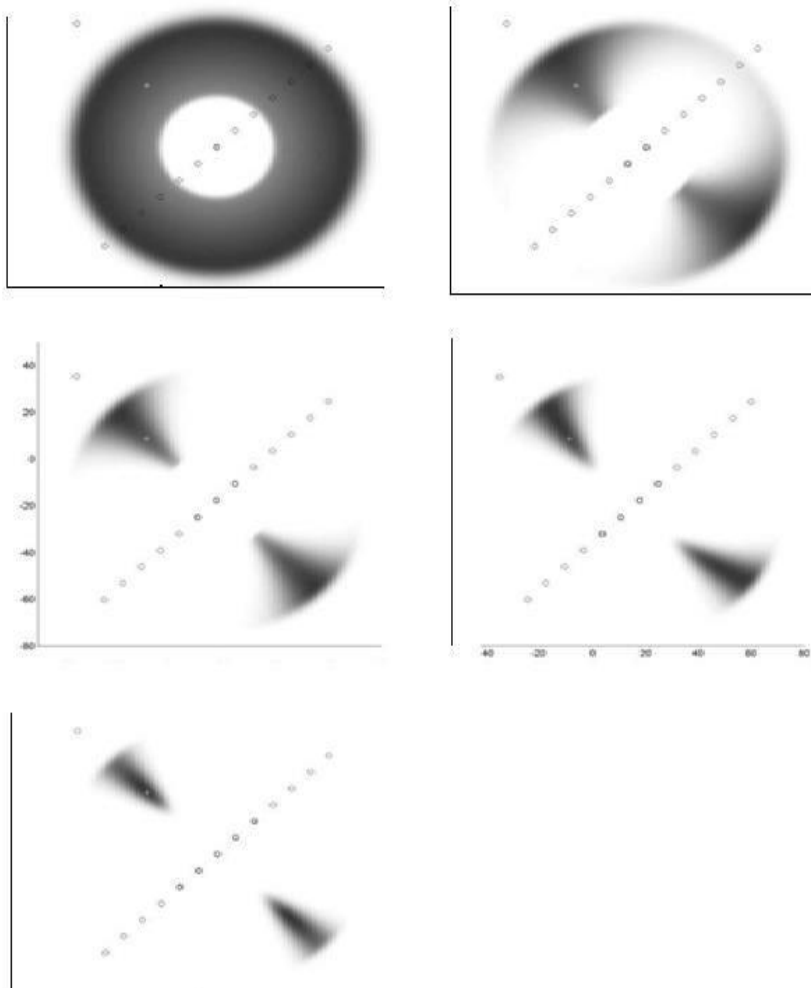
We first consider sensor selection based on a nearest neighborhood (NN) criterion. Using this criterion, the leader node at the center always selects the nearest node among those whose measurements have not been incorporated. Figure 5.5 gives a sequence of snapshots of the localization results based on the NN criterion. Figure 5.5(a) shows the posterior distribution after incorporating the measurement from the initial leader sensor. Next, using the NN criterion, the best sensor is the next nearest neighbor in the linear array, and so forth [Figure 5.5(b)]. Figure 5.5(c) shows the resulting posterior distribution after the leader combines its data with the data from its two nearest neighbors in the linear array. The distribution remains as a bimodal distribution as data from additional sensors in the linear array are incorporated [Figure 5.5(d)-(e)], until the sensor at the upper-left corner of the sensor field is selected.

5.3.2 IDSQ: Information-Driven Sensor Querying

In distributed sensor network systems, we must balance the information contribution of individual sensors against the cost of communicating with them. For example, consider the task of selecting among K sensors with measurements $\{z_i\}_{i=1}^K$. Given the current

belief $p(\mathbf{x} | \{z_i\}_{i \in U})$, where $U \subset \{1, \dots, K\}$ is the subset of sensors whose measurement has already been incorporated, the task is to

Figure 5.5 Sensor selection based on the nearest neighbor method. The estimation task here is to localize a stationary target labeled “+”. Circles denote sensors, and thick circles indicate those whose measurements have already been incorporated. (a) Residual uncertainty after incorporating the data from the leader at the center. (b)-(e) Residual uncertainty after incorporating each additional measurement from a selected sensor.



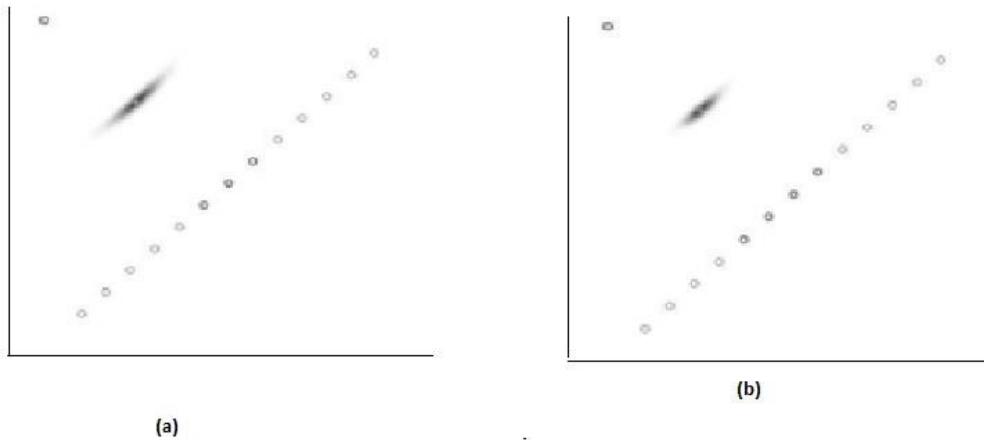


Figure 5.6 Sensor selection based on the Mahalanobis measure of information utility. The localization problem is the same as that in Figure 5.5. The residual uncertainties shown represent the results after incorporating measurements from the 4th and 5th sensors, respectively.

determine which sensor to query among the remaining unincorporated set $A = \{1, \dots, K\} - U$. This is a reasonable strategy for localizing a stationary target. For moving targets, the same sensor may provide informative measurements at different times. The problem of tracking moving targets is discussed in Section 5.4.3.

To be precise, let us define an information utility function

$$\phi: P(\mathbb{R}^d) \rightarrow \mathbb{R}.$$

$P(\mathbb{R}^d)$ represents the class of all probability distributions on d -dimensional state space \mathbb{R}^d for the target state \mathbf{x} . The utility function ϕ assigns a scalar value to each element $p \in P(\mathbb{R}^d)$, which indicates how spread out or uncertain the distribution p is. Smaller values represent a more spread out distribution, while larger values represent a tighter, low-variance distribution. Different choices of ϕ will be discussed later in the section. We further define a cost of obtaining a measurement as a function:

$$\psi: \mathbb{R}^h \rightarrow \mathbb{R}$$

Where \mathbb{R}^h is an h -dimensional measurement space where a measurement vector lies.

In the following, we will refer to sensor l , which holds the current belief, as the *leader node*. The constrained optimization problem of sensor tasking can be reformulated as an unconstrained optimization problem, with the following objective function as a mixture of information utility and cost:

Here φ measures the information utility of incorporating the measurement $\mathbf{z}_j^{(t)}$ from sensor $j \in A$, ψ is the cost of communication and other resources, and γ is the relative weighting of the utility and cost. It should be noted that φ could measure either the total information utility of the belief state after incorporating the new measurement or just the incremental information gain, whichever is easier to compute. With this objective function, the sensor selection criterion takes the form

However, in practice, we do not know the measurement value \mathbf{z}_j without transmitting it to the current aggregation center, the node l , first. Nevertheless, we wish to select the “most likely” best sensor, based on the current belief state $p(\mathbf{x} | \{\mathbf{z}_i\}_{i \in U})$ plus our knowledge of the measurement model and sensor characteristics. For example, the cost function ψ may be estimated as the distance between sensor j and sensor l , or the distance raised to some power, as a rough indicator of how expensive it is to transmit the measurement. As the result,

we often compute an estimate of the cost, $\hat{\psi}$, from parameters such

ψ

In the following, we use the approximations $\hat{\varphi}$ and $\hat{\psi}$ whenever we

φ ψ

discuss utility and cost. Further abusing the notation, the arguments

$\hat{\varphi}$ and $\hat{\psi}$ are not fixed, and the approximation functions may take

φ ψ

various forms, depending on the application and context.

Information Utility Measure I: Mahalanobis Distance

Assume the belief state is well approximated by a Gaussian distribution, and sensor data provide a range estimate, as in acoustic amplitude sensing, for example. In the sensor configuration shown in Figure 5.3, sensor a would provide better information than b because sensor a lies close to the longer axis of the uncertainty ellipsoid and its range constraint would intersect this longer axis transversely. To favor the sensors along the longer axes of an uncertainty ellipsoid, we use the Mahalanobis distance, a distance measure normalized by the uncertainty covariance. The (squared) Mahalanobis distance from \mathbf{y} to μ is defined as

$$(\mathbf{y} - \mu)^T \Sigma^{-1} (\mathbf{y} - \mu).$$

The utility function for a sensor j , with respect to the target position estimate characterized by the mean $\hat{\mathbf{x}}$ and covariance Σ , is defined as the negative of the Mahalanobis distance

$$\varphi(\zeta_j) = -\frac{1}{2} \zeta_j^T \Sigma^{-1} \zeta_j, \quad (5.3)$$

where ζ_j is the position of sensor j .

Intuitively, the points on the $1\text{-}\sigma$ surface of the error covariance ellipsoid are all equidistant from the center under the Mahalanobis measure (Figure 5.7). The utility function works well when the current belief can be well approximated by a Gaussian distribution or

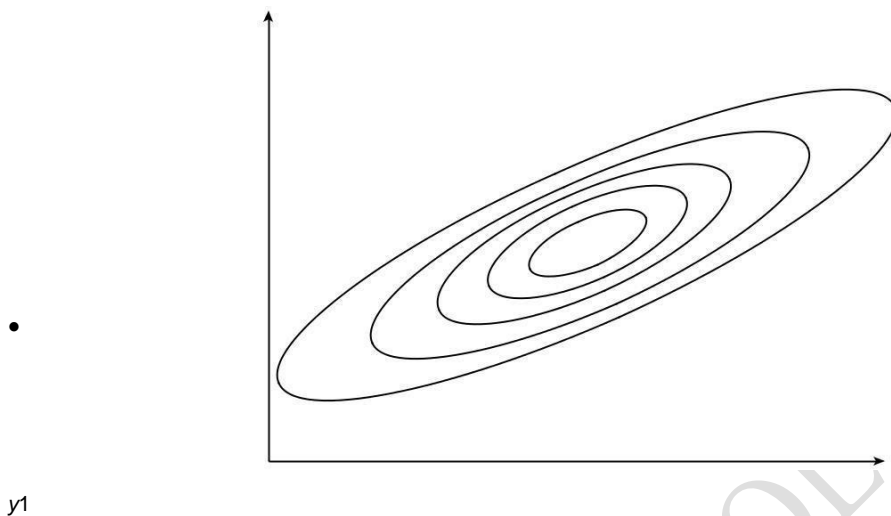


Figure 5.7 Mahalanobis measure: Points on a constant density contour of a Gaussian distribution $N(\mu, \Sigma)$ are equidistant from the mean μ .

the distribution is very elongated, and the sensors are range sensors. However, a bearing sensor reduces the uncertainty along the direction perpendicular to the target bearing and requires a different measure of utility. For a general uncertainty distribution or bearing sensors, we must develop alternative information utility measures.

Information Utility Measure II: Mutual Information

For multimodal, non-Gaussian distributions, a mutual information measure provides a better characterization of the usefulness of sensor data. Additionally, this measure is not limited to information

based only on range data. Assume the current belief is $p(\mathbf{x}^{(t)} | \mathbf{z}^{(t)})$. The contribution of a potential sensor j is measured by

$$I(\mathbf{x}^{(t)}; Z_j | \mathbf{z}^{(t)}) = \int p(\mathbf{x}^{(t)} | \mathbf{z}^{(t)}) \log \frac{p(\mathbf{x}^{(t)} | \mathbf{z}^{(t)}, Z_j)}{p(\mathbf{x}^{(t)} | \mathbf{z}^{(t)})} d\mathbf{x}^{(t)} \quad (5.4)$$

$$\varphi^{(t)} = \int \varphi(\mathbf{x}^{(t)}) \log \frac{\varphi(\mathbf{x}^{(t)})}{\varphi(\mathbf{x}^{(t)}, Z_j)} d\mathbf{x}^{(t)} \quad (5.5)$$

where $I(\cdot ; \cdot)$ measures the mutual information in bits between two random variables. Essentially, maximizing the mutual information is equivalent to selecting a sensor whose measurement $\mathbf{z}_j^{(t+1)}$, when

—
conditioned on the current measurement history $\mathbf{z}^{(t)}$, would provide

the greatest amount of new information about the target location $\mathbf{x}^{(t+1)}$. The mutual information can be interpreted as the Kullback-Leibler divergence between the belief after and before applying the new measurement $\mathbf{z}_j^{(t+1)}$. Therefore, this criterion favors the sensor that, on average, gives the greatest change to the current belief. An implementation of a real-time tracking system using this utility function has shown that this measure is both practically useful and computationally feasible [144].

Appendix C at the end of the book develops additional forms of utility measures. The appropriateness of a particular utility measure for a sensor selection problem depends on two factors: the characteristics of the problem, such as the data and noise models, and the computational complexity of computing the measure. For example, the Mahalanobis measure is easy to compute, although limited to certain data models. The mutual information applies to multimodal distributions, but its computation requires expensive convolution of discrete points if one uses a grid approximation of probability density functions. The choice of which measures to use illustrates the important design trade-off for sensor networks: optimality in information versus feasibility in practical implementation.

5.3.3 Cluster-Leader Based Protocol

The IDSQ method is based on the cluster-leader type of distributed processing protocol. Although the algorithm presented here assumes there is a single belief carrier node active at a time, the basic ideas also apply to scenarios where multiple belief carriers are active simultaneously, as long as the clusters represented by the belief carriers are disjoint from each other; in other words, each sensor senses a single target at a time. Assume we have a cluster of K_1 sensors, each labeled by the integer $\{1, \dots, K_1\}$. A priori, each sensor i only has knowledge of its own position $\zeta_i \in \mathcal{R}^2$. An important prerequisite is the appropriate cluster formation and leader election before applying the algorithm. The cluster may be initially formed from sensors with detections above a threshold and updated as the signal

source moves. A leader may be elected based on relative magnitude of measurement or time of detection (discussed later in this section). Techniques for clustering and leader election are also discussed in Section 4.2 of Chapter 4.

We develop an IDSQ algorithm, using an information criterion for sensor selection and Bayesian filtering for data fusion (see Section 2.2.3), in the context of localization tasks. As pointed out earlier, the basic algorithm introduced here should be equally applicable to other sensing problems. Figure 5.8 shows the flowchart of this algorithm which is identical for every sensor in the cluster. The algorithm works as follows:

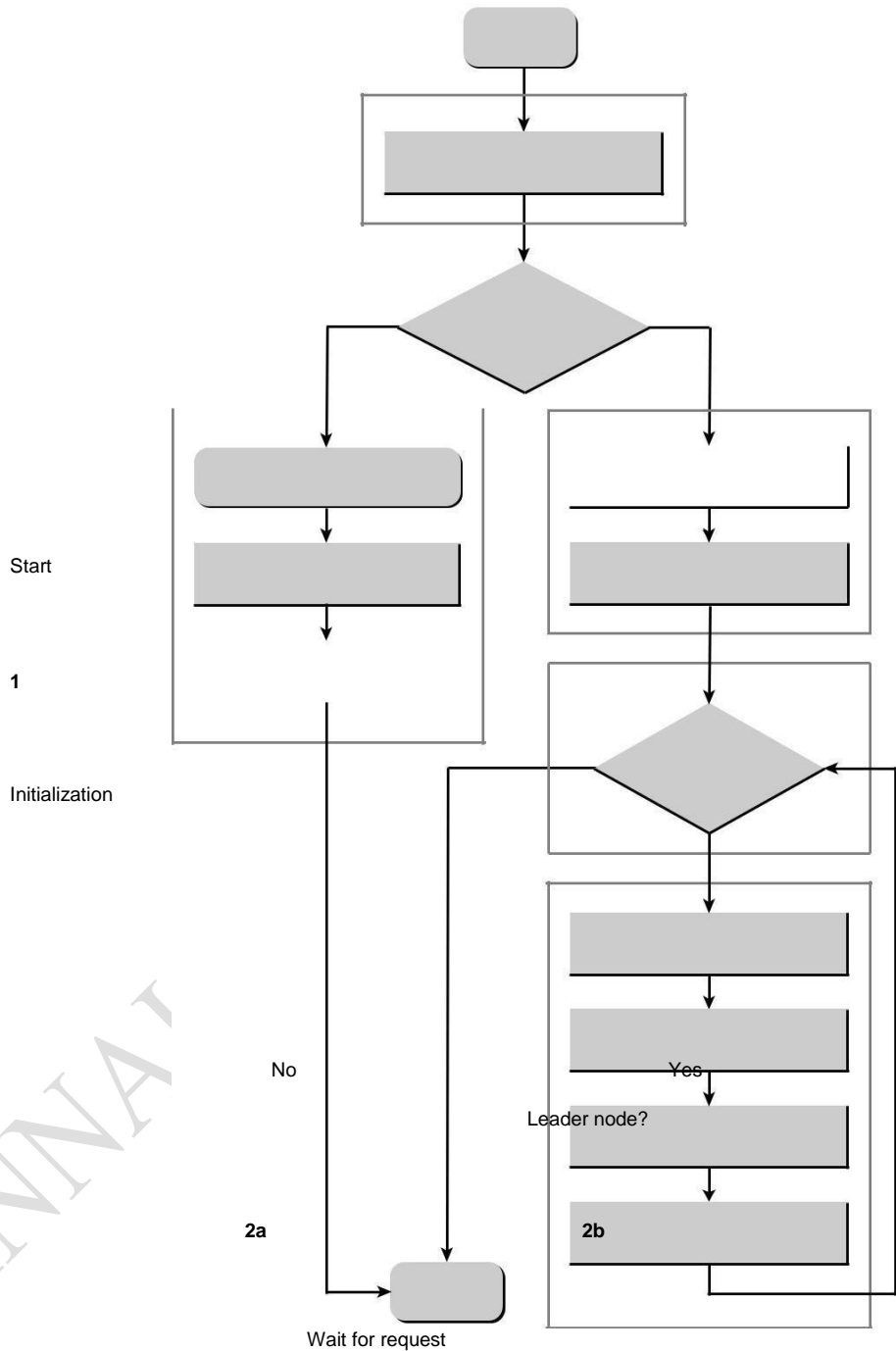
Initialization (Step 1): Each sensor runs an initialization routine through which a leader node is elected from a cluster of K sensors who have detections. The leader election protocol will be considered later. Other sensors in the cluster communicate their own characteristics $\{\lambda_i\}_{i=1}^K$, ($i = l$), as defined in Section 2.2.1, which include the position and noise variance of each sensor, to the leader l .

Follower Nodes (Step 2a): If the sensor node is not the leader, then the algorithm follows the left branch in Figure 5.8. These nodes will wait for the leader node to query them, and if they are queried, they will process their measurements and transmit the queried information back to the leader.

Initial Sensor Reading (Step 2b): If the sensor node is the leader, then the algorithm follows the right branch in Figure 5.8. The leader node will then calculate a representation of the belief state with its own measurement, $p(\mathbf{x} | \mathbf{z}^l)$, and

begin to keep track of which sensor measurements have been incorporated into the belief state, $U = \{l\} \subset \{1, \dots, K\}$.

Again, it is assumed that the leader node has knowledge of the characteristics $\{\lambda_i\}_{i=1}^K$ of all the sensors within the cluster.



Calculate initial belief

Calculate information

and used sensors

3

Send information

Yes

Belief good
enough?

No

4

Sensor selection algorithm

Send information request

Wait for information

Update belief and
used sensors

Finish

Figure 5.8 Flowchart of the information-driven sensor querying (IDSQ) algorithm for each sensor (adapted from [43]).

Belief Quality Test (Step 3): If the belief is good enough, based on some measure of goodness such as the size of belief, the leader node is finished processing. Otherwise, it will continue with sensor selection.

Sensor Selection (Step 4): Based on the belief state, $p(\mathbf{x} | \{\mathbf{z}_i\}_{i \in U})$, and sensor characteristics, $\{\lambda_i\}_{i=1}^K$, pick a sensor node from $\{1, \dots, K\} - U$ that satisfies some information criterion Φ , assuming the one-hop cost is identical for all sensors. Say that node is j . Then, the leader will send a request for sensor j 's measurement, and when the leader receives the requested information, it will

update the belief state with \mathbf{z}_j to get a representation of $p(\mathbf{x} | \{\mathbf{z}_i\}_{i \in U \cup \{j\}})$, and

add j to the set of sensors whose measurements have already been incorporated:

$$:= U \cup \{j\}.$$

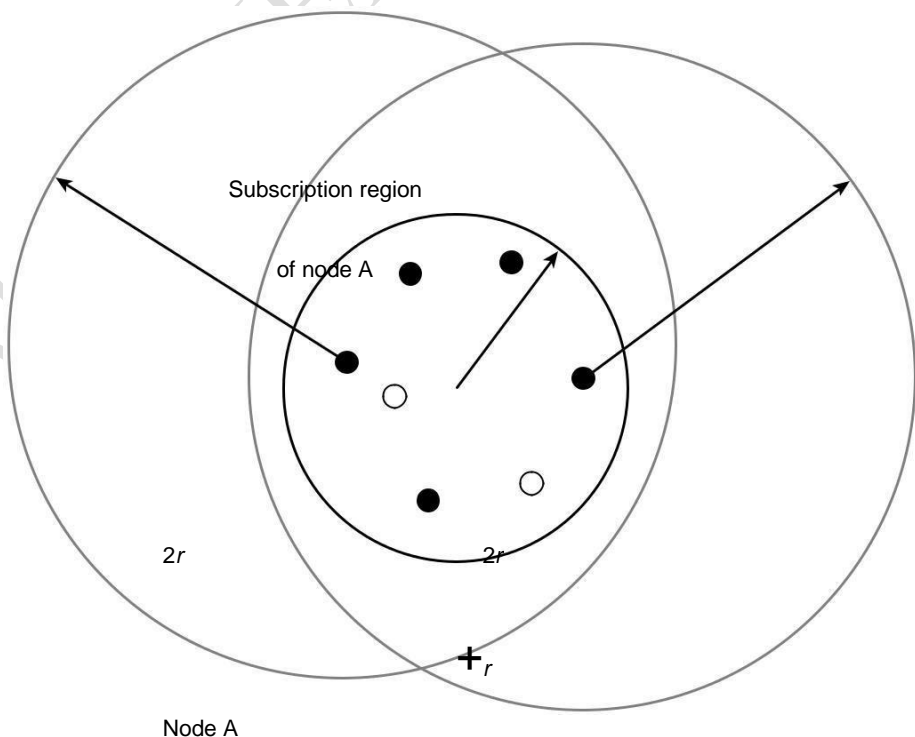
Now, loop back to step 3 until the belief state is good enough.

At the end of this algorithm, the leader node contains all the information about the belief from the sensor nodes by intelligently querying a subset of the nodes that provide the majority of the information. This reduces unnecessary power consumption by transmitting only the most useful information to the leader node. This computation can be thought of as a local computation for this cluster. The belief stored by the leader can then be passed up for processing at higher levels. In steps 2b and 4, some form of representation of the belief $p(\mathbf{x} | \{\mathbf{z}_i\}_{i \in U})$ is stored at the leader node. Considerations for the particular representation of the belief was mentioned in Section 2.3. In step 4, an information criterion is used to select the next sensor. Different measures of information utility, were discussed in Section 5.3.2 and then in details in Appendix C at the end of the book.

Leader Election Protocol

In the leader-based protocol, it is necessary to design efficient and robust algorithms for electing a leader, since typically more than one sensor may have detections about a target simultaneously. Here, we describe a geographically-based leader election scheme that resolves contention and elects a single leader via message exchange.

First, consider the ideal situation. If the signal of a target propagates isotropically and attenuates monotonically with distance, the sensors physically closer to the target are more likely to detect the target than the sensors far away. One can compute an “alarm region,” similar to a $3\text{-}\sigma$ region of a Gaussian distribution, such that most (e.g., 99 percent) of the sensors with detections fall in the region. This is illustrated in Figure 5.9. Sensor nodes are marked with small circles; the dark ones have detected a target. Assume the target is located at \mathbf{x} (marked with a “+” in the figure), the alarm region is a



Alarm Region

Subscription region

of node B

Figure 5.9 Leader election in a detection region (adapted from [142]).

5.3 Information-Based Sensor Tasking **157**

disk centered at \mathbf{x} with some radius r , where r is determined by the observation model. In practice, we choose r based on the maximum detection range plus a moderate amount of margin to account for possible target motion during a sample period.

Ideally, nodes in an alarm region should collaborate to resolve their contention and elect a single leader from that region. However, the exact location of the alarm region is unknown since the target position \mathbf{x} is unknown. Each node with a detection only knows that the target is within a distance of r , and a possible competitor could be a farther distance r from the target. Thus in the absence of a “message center,” a node notifies all nodes within a radius $2r$ of itself (the potential “competitors” for leadership) of its detection.

Upon detection, each node broadcasts to all nodes in the enlarged alarm region a DETECTION message containing a time stamp indicating when the detection is declared, and the likelihood ratio $p(\mathbf{z}|H1)/p(\mathbf{z}|H0)$, where $H1$ or $H0$ denote the hypotheses of the target being present or not. The higher this ratio, the more confident the detecting node is of its detection. We rely on a clock synchronization algorithm to make all time stamps comparable, as discussed in Section 4.3. We also need a routing mechanism to effectively limit the

propagation of the detection messages to the specified region, using, for example, the geographical routing to a region method presented in Section 3.4.4.

After sending out its own detection message, the node checks all detection packets received within an interval of t_{comm} . The value of t_{comm} should be long enough for all messages to reach their destination, yet not too long so that the target can be considered approximately stationary. These messages are then compared with the node's own detection. The node winning this election then becomes the leader immediately, with no need for further confirmation. The election procedure is as follows:

If none of the messages is time-stamped earlier than the node's own detection, the node declares itself leader.

If there are one or more messages with an earlier time stamp, the node knows that it is not the leader.

Chapter 5 Sensor Tasking and Control

If none of the messages contains earlier time stamps, but some message contains a time stamp identical to the node's detection time, the node compares the likelihood ratio. If the node's likelihood ratio is higher, the node becomes the leader.

This algorithm elects only one leader per target in an ideal situation. However, in other situations multiple leaders may result. For example, if the DETECTION packet with the earliest detection time stamp fails to reach all the destination nodes, multiple nodes may find that they are the "earliest" detection and each may initiate a localization task. One way to consolidate the multiple leaders is to follow the initial election with another round of election, this time involving only the elected leaders from the initial round. Since there are fewer nodes to send out messages, the probability of DETECTION packets reaching every leader is greatly increased.

Simulation Experiments

The leader-based protocol is applied to the problem of spatial localization of a stationary target based on amplitude measurements from a network of 14 sensors, as arranged in Figure 5.4. The goal is to compare different sensor selection criteria and validate the IDSQ algo-

rithm presented earlier. Assuming acoustic sensors, the measurement model for each sensor i is given in Equation (2.4) of Chapter 2. For the experiment, we further assume $a \in \mathcal{R}$, the source amplitude of the target signal, is uniformly distributed in the interval $[a_{low}, a_{high}]$. For simplicity, in the simulation examples considered in this section, a leader is chosen to be the one whose position ζ_l is closest to the centroid of the sensors, that is,

$$l = \arg \min_{j \in \{1, \dots, N\}} \sum_{i=1}^N \|\zeta_j - \zeta_i\|^2$$

We test four different criteria for choosing the next sensor:

A. Nearest neighbor

3 Information-Based Sensor Tasking **159**

B. Mahalanobis distance

C. Maximum likelihood

D. Best feasible region (or ground truth for one-step optimization)

Criterion D is uncomputable in practice since it requires knowledge of sensor measurement values in order to determine which sensor to use. However, we use it as a basis for comparison with the other criteria. These four criteria are defined more precisely in Appendix D at the end of the book.

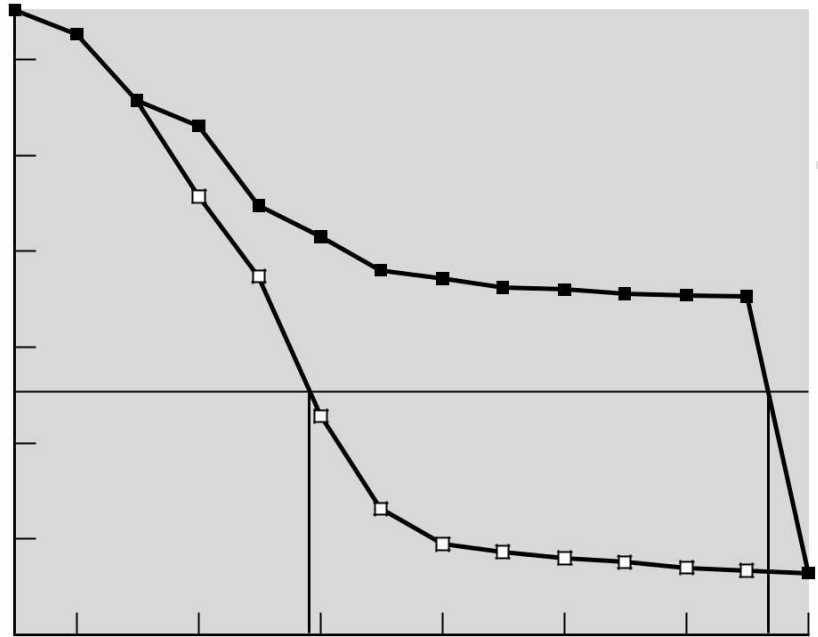
Additional details of the protocol and sensor selection criteria are specified as follows. For the simulations, we have chosen $alow = 10$

and $ahigh = 50$. The sensor noise variance σ_i is set to 0.1, which is about 10 percent of the signal amplitude when the amplitude of the target

is 30 and the target is at a distance of 30 units from the sensor. The parameter β in criterion D (as specified in Appendix D) is chosen to be 2, since this value covers 99 percent of all possible noise instances. For the first simulation, the signal attenuation exponent α is set to 1.6, which considers reflections from the ground surface. Then α is set to 2 for the second simulation, which is the attenuation exponent in free space with no reflections or absorption. The shape of the uncertainty region is sensitive to different choices of α ; however, the comparative performance of the sensor selection algorithm for different selection criteria turns out to be relatively independent of α .

The first simulation is a pedagogical example to illustrate the use-fulness of incorporating a sensor selection algorithm into the sensor network. Figure 5.4 earlier in the chapter shows the layout of 14 microphones. The one microphone not in the linear array is placed so that it is farther from the leader node than the farthest micro-phon in the linear array. With sensor measurements generated by a stationary source in the middle of the sensor network, sensor selection criteria A and B are compared. The difference in the two criteria is the order in which the sensors' measurements are incorporated into the belief.

Figure 5.10 shows a plot of the logarithm of the determinant of the error covariance of the belief state (or the volume of the error Chapter 5 Sensor Tasking and Control



ANNAL WOMEN

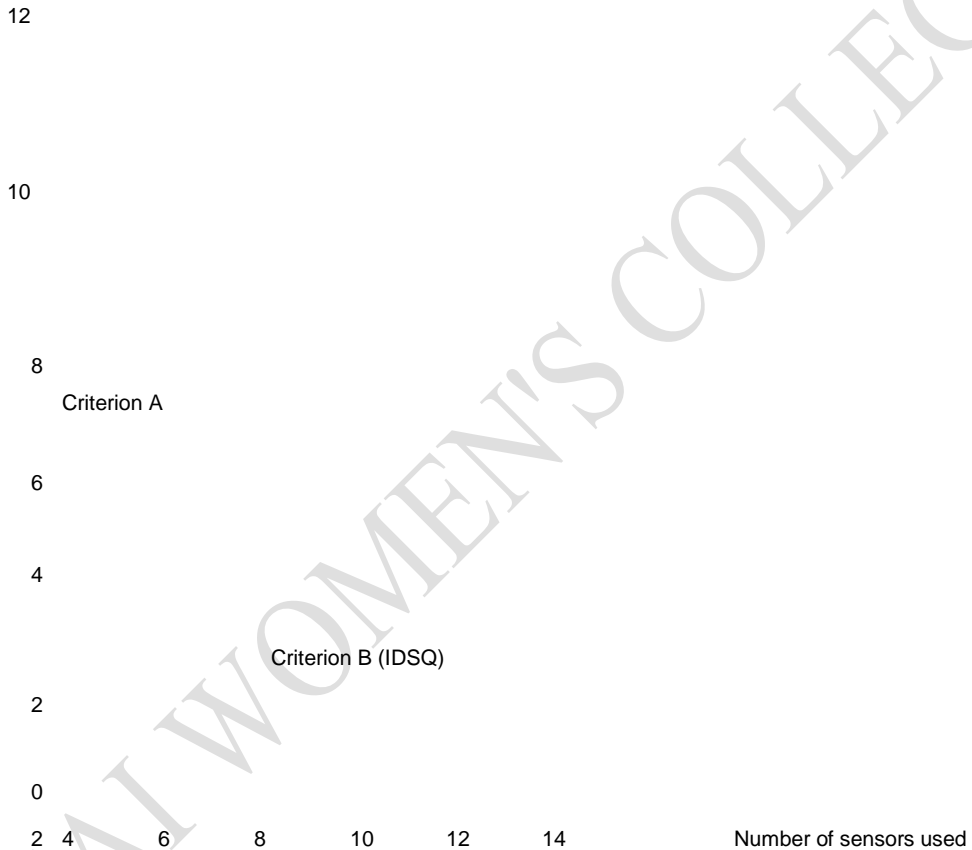


figure 5.10 Determinant of the error covariance for selection criteria A and B for the sensor layout shown in Figure 5.4. Criterion A tasks 14 sensors, while B tasks 6 sensors to be below an error threshold of five units (adapted from [43]).

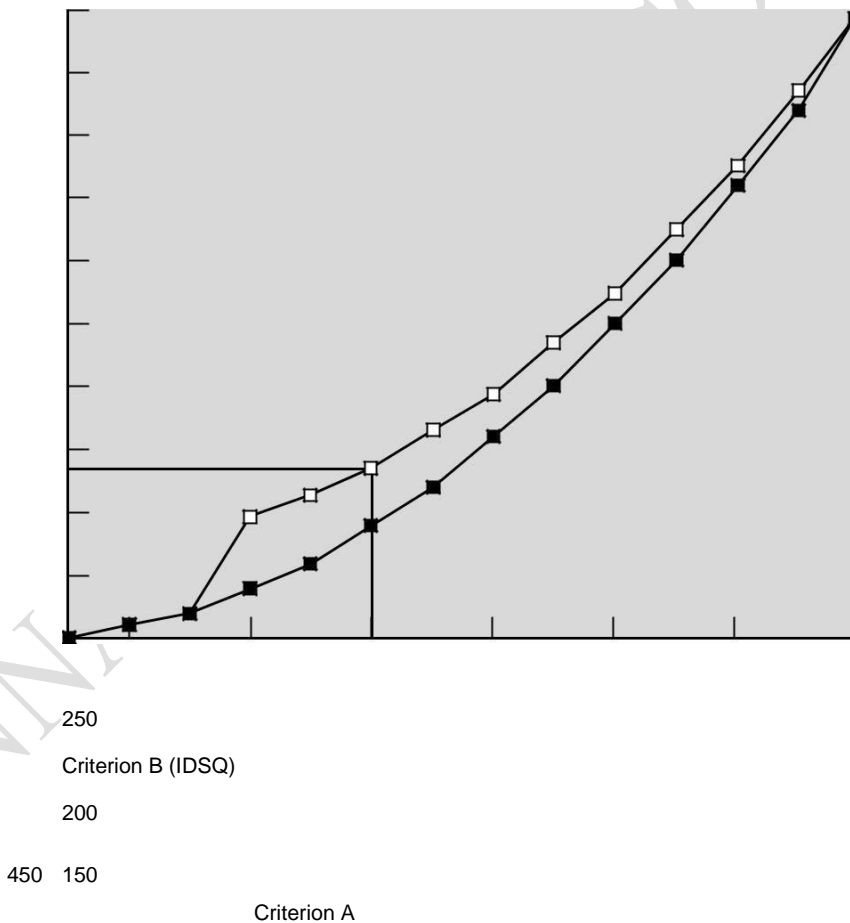
ellipsoid) versus the number of sensors incorporated. Indeed, the volume of the error covariance under selection criterion B is less than the volume of the error covariance under selection criterion A for the same number of sensors, after more than three sensors are

selected. When all 14 sensors have been accounted for, both methods produce the same amount of error reduction.

A plot of the communication distance versus the number of sensors incorporated is shown in Figure 5.11. Certainly, the curve for selection criterion A is the lower bound for any other criterion. Criterion A optimizes the network to use the minimum amount of communication energy when incorporating sensor information; however, it largely ignores the information content of these sensors.

5.3 Information-Based Sensor Tasking

161



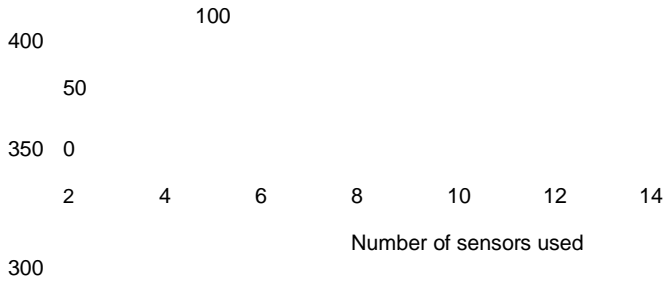


Figure 5.11 Total communication distance versus the number of sensors queried for selection criteria A and B for the sensor layout shown in Figure 5.4. For achieving the same threshold of the error, A tasks 14 sensors and uses nearly 500 units of communication distance, whereas B tasks 6 sensors and uses less than 150 units of communication distance (adapted from [43]).

A more informative interpretation of the figure is to compare the amount of energy it takes for criterion A and criterion B to achieve the same level of accuracy. Examining Figure 5.10, we see that under criterion A, in order for the log determinant of the covariance value to be less than 5, criterion A requires all 14 sensors to be tasked. On the other hand, criterion B requires only 6 sensors to be tasked. Now, comparing the total communication distance for this level of accuracy from Figure 5.11, we see that criterion B requires less than 150 units of communication distance for tasking 6 sensors, as opposed to nearly 500 units of communication distance for tasking

Chapter 5 Sensor Tasking and Control

all 14 sensors. Indeed, for a given level of accuracy, B generally requires less communication distance than A.

The above simulation was carried out on a specific layout of the sensors, and the striking improvement of the error was largely due to the fact that most of the sensors were in a linear array. Thus, the next simulation will explore which one does better, on average, with randomly placed sensors.

Microphones are placed uniformly in a given square region as shown in Figure 5.12(a). The target is placed in the middle of the square region and given a random amplitude. Then, the sensor algorithm for each of the different sensor selection criteria described earlier is run for 200 runs. Figure 5.12(b) shows a comparison between selection criteria A and B. There are three segments in each bar. The bottom segment represents the percentage of runs in which the error for B is strictly less than the error for A after k sensors have been incorporated. The middle represents a tie. The upper segment represents the percentage of runs in which the error for B is larger than the error for A. Since the bottom segment is larger than the upper one (except for the initial and final phases when they are tied), this shows B performs better than A on average.

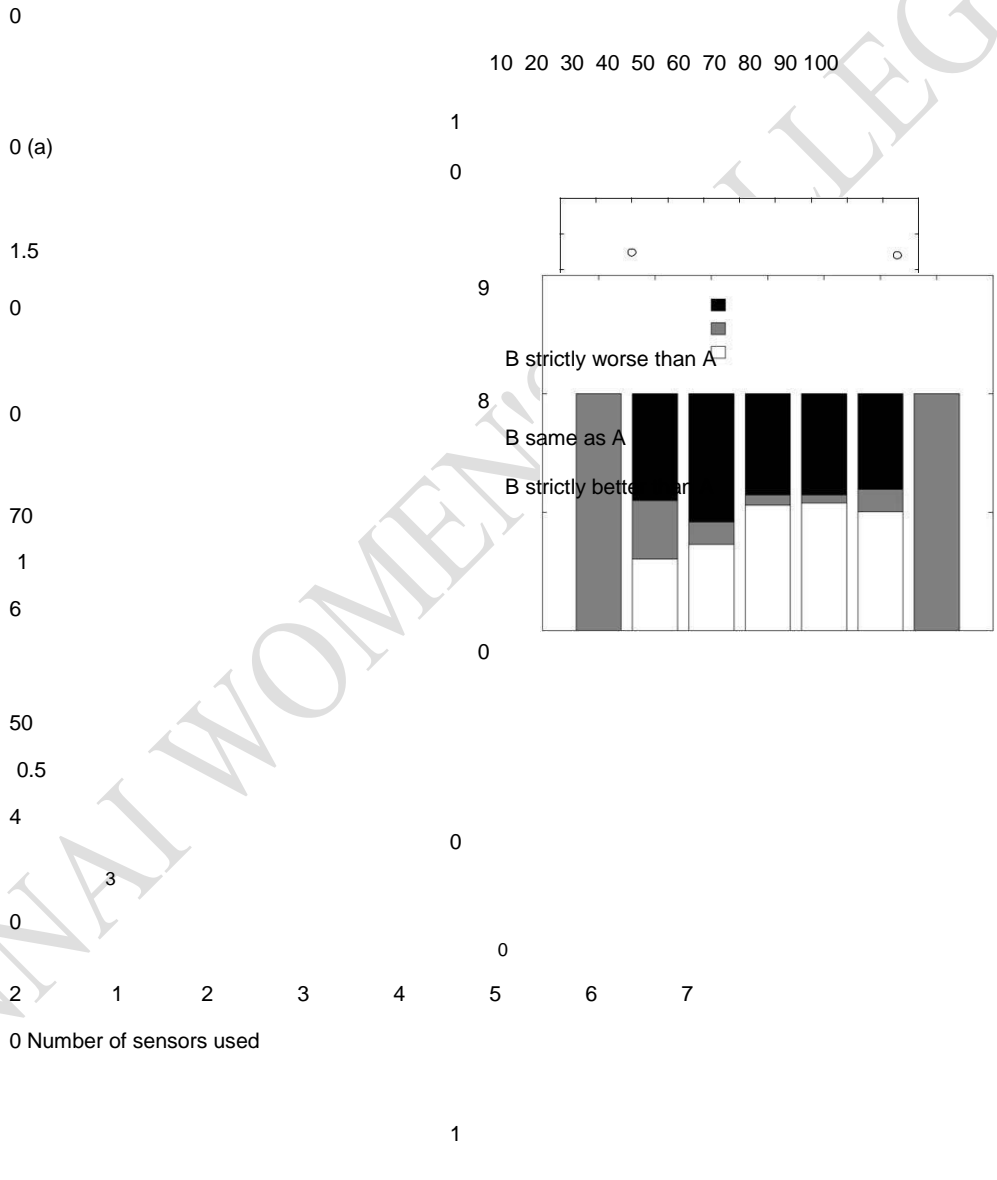
Figures 5.12(c) and (d) show comparisons of sensor criteria C and D versus B. The performance of C is comparable to B and, as expected, D is better than B on average. The reason D is not always better than B over a set of sensors is because both are greedy criteria. The n^{th} best sensor is chosen incrementally with the first $n - 1$ sensors already fixed. Fixing the previous $n - 1$ sensors when choosing the n^{th} sensor is certainly suboptimal to choosing n sensors all at once to maximize the information content of the belief.

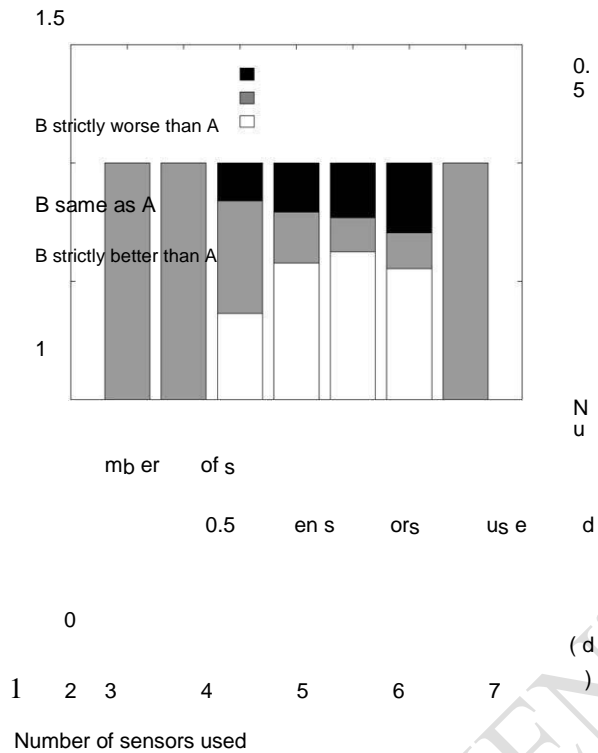
5.3.4 Sensor Tasking in Tracking Relations

The focus of this section is sensor tasking and control in the context of tracking spatial or temporal relations between objects and local or global attributes of the environment—as

opposed to the detailed estimation of positions and poses of individual objects. In certain cases, high-level behaviors of objects may be trackable more robustly than their exact positions, relations between objects may be directly

5.3 Information-Based Sensor Tasking 163





(b)

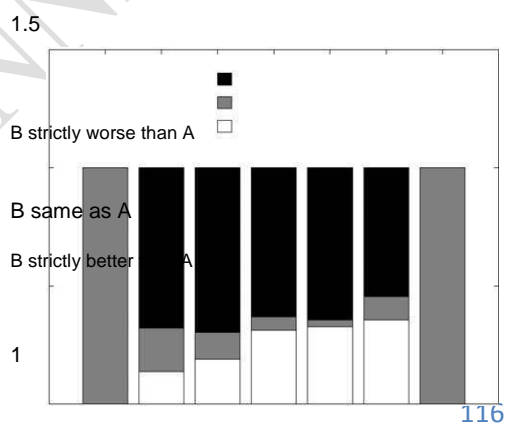
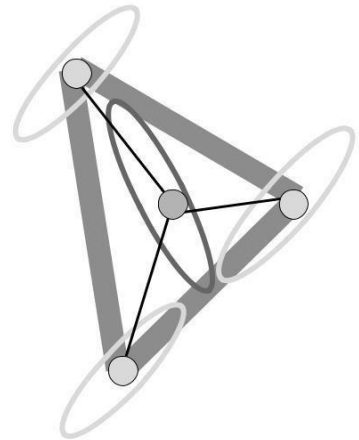
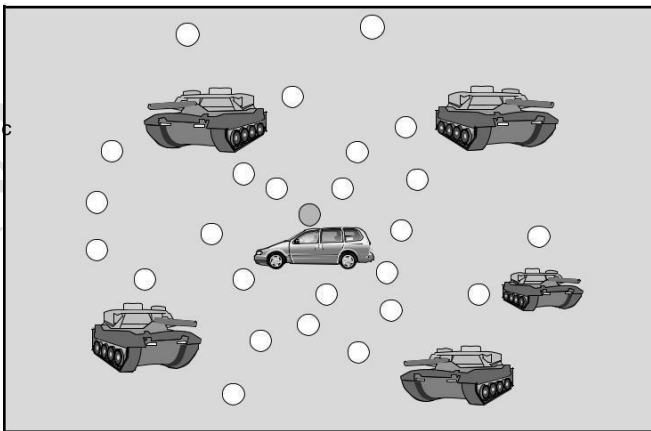


Figure 5.12 (a) Layout of seven randomly placed sensors (circles) with the target (cross) in the middle. (b) Percentage of runs where B performs better than A for seven randomly placed sensors. (c) Percentage of runs where B performs better than C for seven randomly placed sensors. (d) Percentage of runs where B performs better than D for seven randomly placed sensors (adapted from [43]).

observable by sensors, and the large-scale behavior of an ensemble of objects may be easier to ascertain than the motion of the individual objects. By focusing on relations and the logical structure of the evidence with respect to the task at hand, information-based approaches will be able to allocate sensing, computation, and communication resources where they are most needed.

An example of tracking relations is the “Am I surrounded?” problem: determine if a friendly vehicle is surrounded by a number of enemy tanks [Figure 5.13(a)]. The goal is to design a sensing strategy that extracts global relations among the vehicles in question without first having to solve local problems, such as accurately localizing individual vehicles. One definition of whether the friendly vehicle is surrounded by the tanks is to test if the vehicle is inside the geometric convex hull formed by the enemy tanks. Although the notion of “Am I surrounded?” is somewhat application dependent, we will use this definition to show how such a global relation can be determined by tasking appropriate sensors, based on how their local sensing actions can contribute to the assertion of the relation. We start by decomposing a global relation into more primitive ones. For example, the global relation of whether a point d is surrounded by points a, b, c can be established by the conjunction of the more primitive relations $CCW(a, b, d)$, $CCW(b, c, d)$, $CCW(c, a, d)$, where CCW denotes the counterclockwise relation [Figure 5.13(b)].

To establish a CCW relation among three objects, sensors are selected to localize the objects, with the objective of maximizing



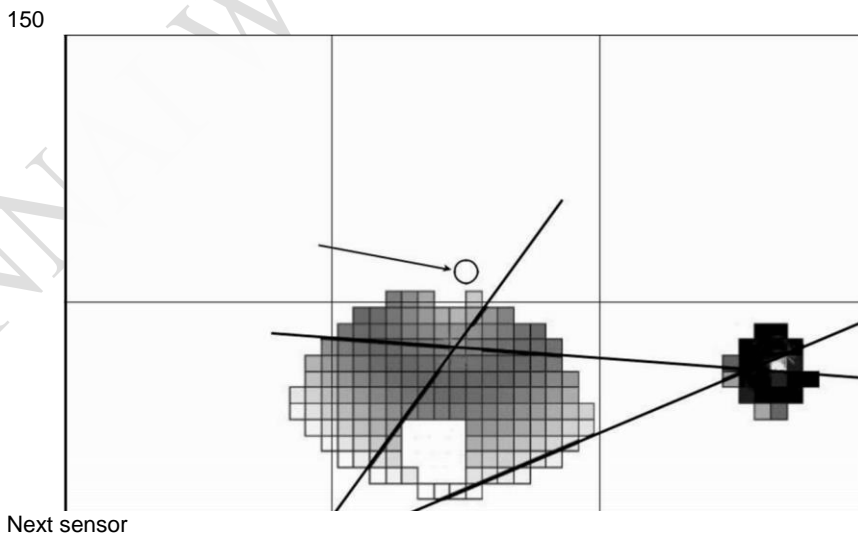
d b

a

(a) **re 5.13** (a) A global relation of “Am I surrounded?” Here, a friendly vehicle in the middle is attempting to determine if it is inside the convex hull of enemy tanks.

(b) Decomposition of a global relation into three more primitive CCW relations.

the reduction in the uncertainty of the CCW relation while minimizing the number of sensor queries. To resolve a CCW relation, a sensor with maximum expected reduction in the uncertainty of the CCW relation is chosen. For example, to resolve the $CCW(a, b, d)$ in Figure 5.13(b), where ellipses denote uncertainty covariances for localization of a , b , and d , one notices that tasking sensors that minimize the error for node d can result in faster resolution of the CCW relation. Figure 5.14 shows a simulation of the CCW resolution as



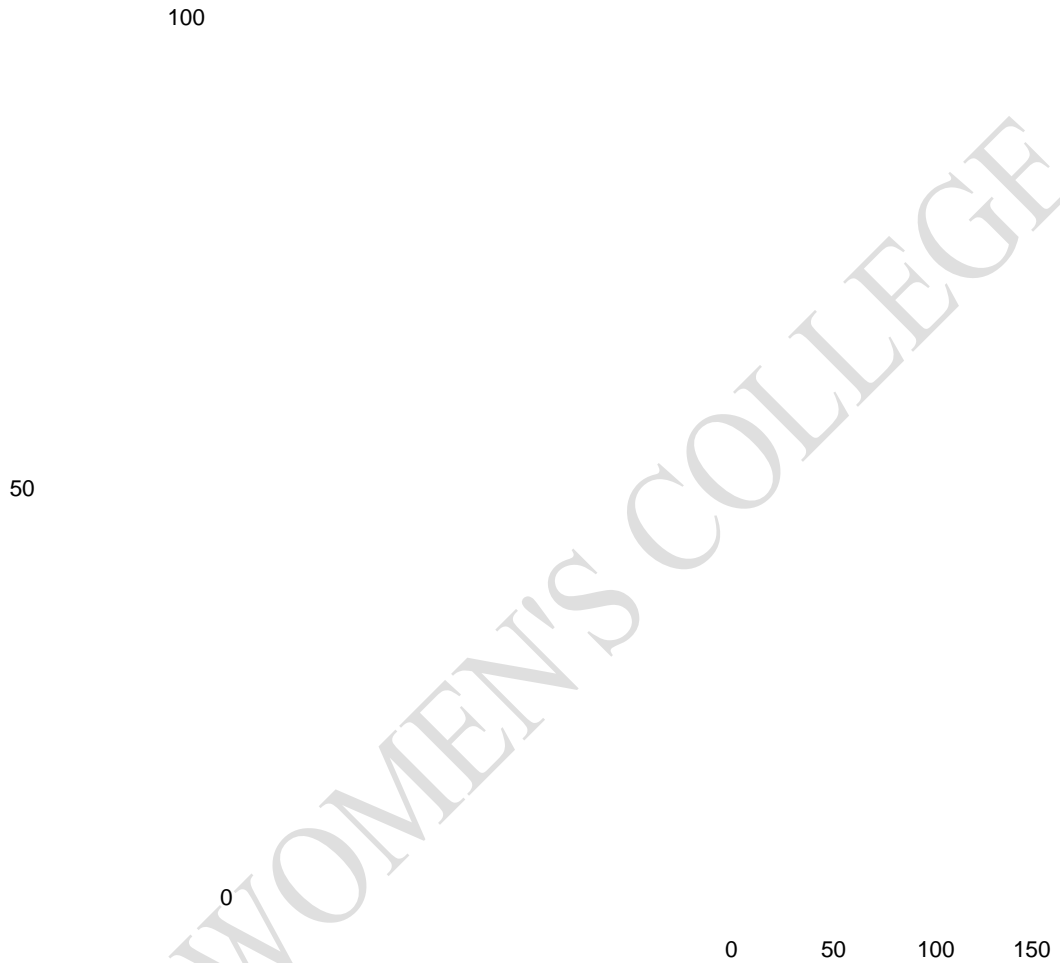


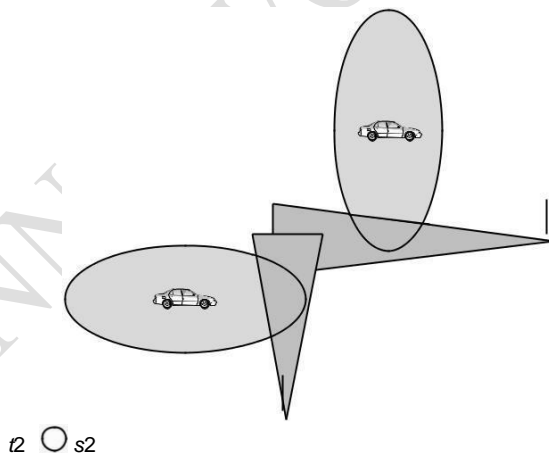
Figure 5.14 A CCW relation is being resolved by tasking sensors with maximum expected information gain. A “+” denotes the true location for each target (adapted from [87]).

Chapter 5 Sensor Tasking and Control

sensors are selected. In the figure, each node is localized with a set of range sensors, resulting in residual uncertainties about object positions shown in the figure.

Note that uncertainty in the CCW relation is caused by possible collinearity or nearcollinearity of the targets. The best sensor selection strategy for removing collinearity uncertainty may be quite different from the best strategy for localizing the sensors, irrespective of the CCW relation. Consider, for instance, the situation in Figure 5.15. Note that a passive infrared (PIR) sensor s_1 may look at one of the tails of the distribution for target t_3 and, upon seeing nothing there, lop off a large chunk of this distribution and reduce its spread. Yet that reduction is almost useless as far as eliminating wrongly oriented triplets of possible target locations. Another PIR sensor s_2 may lop off a smaller part of the t_2 distribution, yet have

much more significant benefit toward certifying CCW(t_1, t_2, t_3). Analogous to the sensor selection for the point localization problems, we need to develop a model of utility and cost that relates the resolution of global relations to the sensing and communications required. This remains as an open problem for future research.



○ s1

figure 5.15 The effect on CCW of alternate sensor readings (adapted from [87]).



5.4 Joint Routing and Information Aggregation

A primary purpose of sensing in a sensor network is to collect and aggregate information about a phenomenon of interest. While IDSQ provides us with a method of selecting the optimal order of sensors to obtain maximum incremental information gain, it does not specifically define how a query is routed through the network or the information is gathered along the routing path. This section outlines a number of techniques that exploit the composite objective function (5.1) to dynamically determine the optimal routing path.

Consider the following two scenarios in which an information aggregation query is injected into the network. The first one is illustrated in Figure 5.16(a). A user (which may be another sensor node)

Exit

(a) (b)

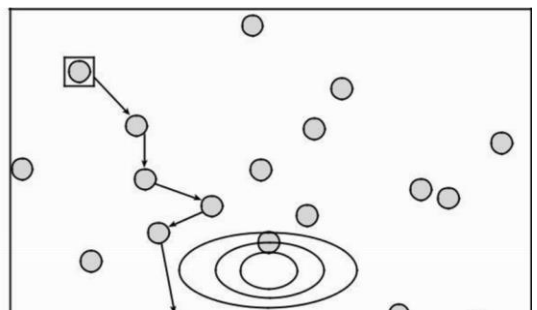
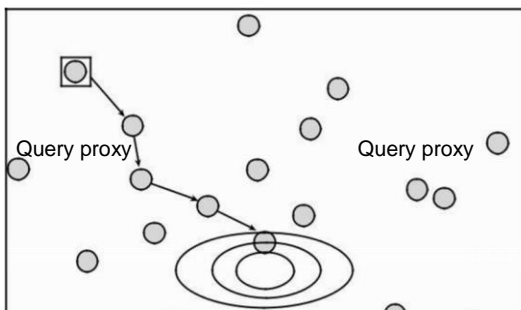


Figure 5.16 Routing and information aggregation scenarios: (a) Routing from a query proxy to the high activity region and back. The ellipses represent iso-contours of an information field, which is maximal at the center. The goal of routing is to maximally aggregate information along a path while keeping the cost minimal. (b) Routing from a query proxy to an exit node, maximizing information gain along the path (adapted from [145]).

issues a query from an arbitrary node, which we call a *query proxy node*, requesting the sensor network to collect information about a phenomenon of interest. The query proxy has to figure out where such information can be collected and routes the query toward the high information content region. This differs from routing in communication networks where the destination is often known a priori to the sender. Here, the destination is unknown and is dynamically estimated by the network's knowledge about the physical phenomenon.

The second routing scenario is pictured in Figure 5.16(b). A user—for example, a police officer—may issue a query to a node, asking the sensor network to collect information and report the result to an extraction or exit node—for example, a police station—where the information can be extracted for further processing. In this scenario, the query proxy and exit nodes may be far away from the high information content region. A path taking a detour toward the high information region may be more preferable than the shortest path. In this section, we will first consider the sensor tasking and routing problem for the first scenario. The problems associated with the second scenario are studied in Section 5.4.2.

5.4.1 Moving Center of Aggregation

We have described a class of algorithms based on the fixed belief carrier protocol in which a designated node such as a cluster leader holds the belief state. In that case, the querying node selects optimal sensors to request data from, using the information utility measures. For example, using the Mahalanobis distance measure, the querying node can determine which node can provide the most useful information while balancing the communication cost, without the need to obtain the remote sensor data first. We now consider a dynamic belief carrier protocol in which the belief is successively handed off to sensor nodes closest to locations where “useful” sensor data are being generated. In the dynamic case, the current sensor node updates the belief with its measurement and sends the estimation to the next neighbor that it determines can best improve the estimation.

Locally Optimal Search

Here the information query is directed by local decisions of individual sensor nodes and guided into regions maximizing the objective function J as defined in (5.1). Note that the function J is incrementally updated along with the belief updates along the routing path. The local decisions can be based on different criteria:

For each sensor k that makes the current routing decision, evaluate the objective function J at the positions of the m sensors within a local neighborhood determined by the communication distance, and pick the sensor j that maximizes the objective function locally within the neighborhood:

$$\hat{j} = \arg \max_j (J(\zeta_j)), \quad \forall k$$

where ζ_j is the position of the node j .

Choose the next routing sensor in the direction of the gradient of the objective function, ∇J .

Among all sensors within the local communication neighborhood, choose the sensor j such that

$$\hat{j} = \arg \max_j \frac{(\nabla J)^T \cdot (\zeta_j - \zeta_k)}{\|\zeta_j - \zeta_k\|},$$

$$\nabla \cdot \zeta_j - \zeta_k$$

where ζ_k is the position of the current routing node, and “ \cdot ” denotes the inner product of two vectors.

If the network routing layer supports geographical routing, as described in Section 3.4, the querying sensor can directly route

the query to the sensor closest to the optimum position. The optimum position ζ_o corresponds to the location where the utility function φ is maximized and can be computed by the querying sensor by evaluating the utility function:

$$\zeta_o = \arg\zeta [\nabla\varphi = 0] . \quad (5.5)$$

Chapter 5 Sensor Tasking and Control

However, the destination is optimal only with respect to the current data the querying sensor has. As the query travels through the sensor nodes in the network, additional sensor data may be incrementally combined to continuously update the optimum position.

Instead of following the local gradients of the objective function throughout the routing path, the chosen direction at any hop can

be biased toward the direction aiming at the optimum position, ζ_o . This variation of the gradient ascent algorithm is most useful in regions of small gradients of the objective function, that is, where the objective function is relatively flat. The direction toward the maximum of the objective function can be found by evaluating

(5.5) at any routing step. This allows a node to compute locally the direction toward the optimum position ($\zeta_o - \zeta_k$), where ζ_k denotes the position of the current routing sensor. The optimal direction toward the next sensor can be chosen according to a weighted average of the gradient of the objective function and the direct connection between the current sensor and the optimum position:

$$= \beta \nabla J + (1 - \beta) (\zeta_o - \zeta_k),$$

where the parameter β can be chosen, for example, as a function of the inverse of the distance between the current and the optimum sensor positions: $\beta = \beta_0 \|\zeta_o - \zeta_k\|^{-1}$. This routing mechanism allows adapting the routing direction to the distance from the optimum position. For small distances, it might be better to follow the gradient of the objective function for the steepest ascent, that is, the fastest information gain. For large distances from the optimum position where the objective function is flat and data is noisy, it is faster to directly go toward the maximum than to follow the gradient ascent.

In order to locally evaluate the objective function and its derivatives, the query needs to be transmitted together with information on the current belief state. This information should be a compact

5.4 Joint Routing and Information Aggregation

171

representation of the current estimate and its uncertainty and must provide complete information to incrementally update the belief state given local sensor measurements. For the earlier example of quantifying the information utility by the Mahalanobis distance, we

need to transmit the triplet $(\mathbf{x}, \mathbf{P}, \zeta)$ with the query, where \mathbf{x} is the

position of the querying sensor, \mathbf{x}^* is the current estimate of the target

position, and \mathbf{P} is the current estimate of the position uncertainty

covariance.

The routing mechanism described earlier can be used to establish a routing path toward the potentially best sensor, along which the measurement from the sensor closest to the optimum position is shipped back. When global knowledge about optimum sensor positions is available, the routing path is optimal whereas information gathering may not be. In the case of local sensor knowledge, the path is only locally optimal because the routing algorithm is a greedy method. The estimate and the estimation uncertainty can be dynamically updated along the routing path. The measurement can also be shipped back to the query-originating node. Since the information utility objective function along the path is monotonically increasing, the information provided by subsequent sensors is getting incrementally better toward the global optimum. When the information is continuously shipped back to the querying sensor, the information arriving in sequential order provides an incremental improvement to the estimate. Once a predefined estimation accuracy is reached, the querying sensor can terminate the query even if it has not yet reached the optimum sensor position. Alternatively, instead of shipping information back to the querying sensor, the result could be read out from the network at the sensor where the information resides.

Simulation Experiments

The objective function used in the greedy routing experiments is chosen according to (5.1), with the information utility and cost terms defined, respectively, as:

$$\varphi(\zeta, \mathbf{x}) = (\zeta^T \mathbf{x})^{-1} (\zeta^T \mathbf{x}), \quad (5.6) \quad j^{\wedge} = -j - \hat{j} - \hat{j}$$

Chapter 5 Sensor Tasking and Control

$$\psi(\zeta_j, \zeta_l) = (\zeta_j - \zeta_l)^T (\zeta_j - \zeta_l), \quad (5.7)$$

where ζ_l represents the position of the querying sensor l .

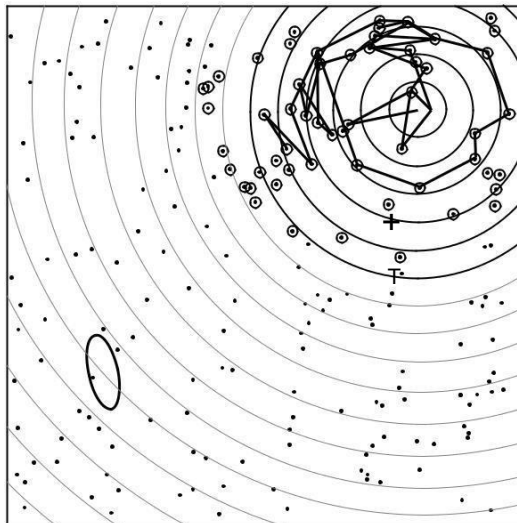
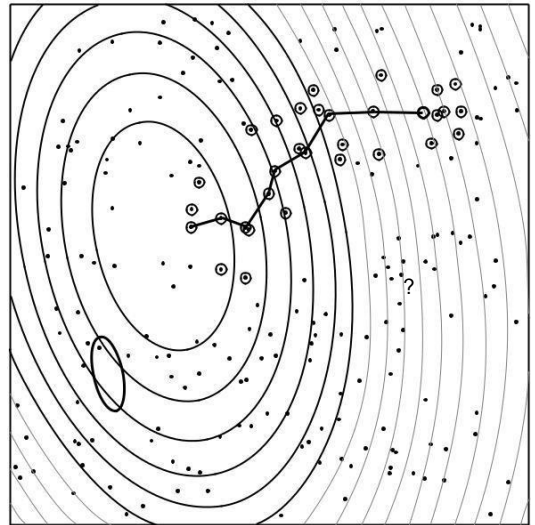
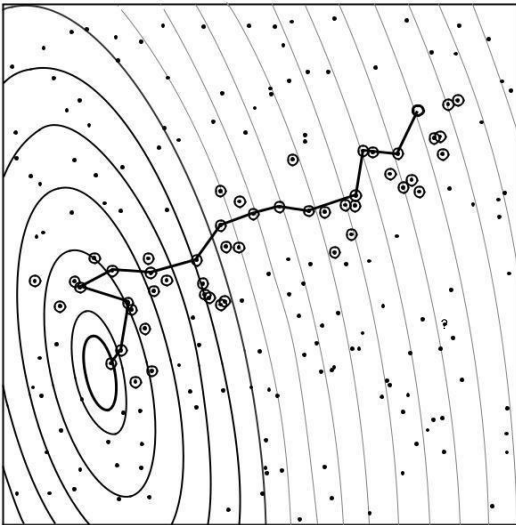
Figure 5.17 shows snapshots of numerical simulations of the greedy routing algorithm on networks of randomly placed sensors. To simplify the illustration, current target position, \mathbf{x}^{\wedge} , and its uncertainty, Σ , were arbitrarily chosen and remained fixed for the run—that is, no incremental update of the belief state was implemented. The value of the objective function across the sensor network is shown as a contour plot, with peaks of the objective function located at the center of ellipses. The circle indicated by a question mark (?) depicts the position of the querying sensor (query origin), and the circle indicated by T depicts the estimated target position, \mathbf{x}^{\wedge} . The current uncertainty in the position estimate, Σ , is depicted as a 90-percentile ellipsoid enclosing the position T.

The goal of the greedy routing algorithm is to guide the query as close as possible toward the maximum of the objective function, following the local gradients to maximize incremental information gain. While the case of trade-off parameter $\gamma = 1$ represents maximum information gain, ignoring the distance from the querying sensor (and hence the energy cost), the case $\gamma = 0$ minimizes the energy cost, ignoring the information gain. For other choices of $0 < \gamma < 1$, the composite objective function represents a trade-off between information gain and energy cost.

Figure 5.17 shows how variation of the trade-off parameter γ morphs the shape of the objective function. As γ decreases from 1 to 0, the peak location moves from being centered at the predicted target position ($\gamma = 1$) to the position of the querying sensor ($\gamma = 0$); at the same time, the contours change from being elongated, shaped according to the uncertainty ellipsoid represented by the estimated covariance Σ , toward isotropic. Another interesting aspect of the combined objective function is that the spatial position of its maximum does

not shift linearly between the estimated target position T and the query origin $?$, with varying γ . This can be observed in the case of $\gamma = 0.2$, where the maximum is located off the line connecting T and $?$.

(a) (b)



+
T

oo?

+

• T

(c)

Figure 5.17 Locally optimal routing for $N = 200$ randomly placed sensors, with varying information versus cost tradeoff parameter γ . From (a) to (c): $\gamma = 1$, $\gamma = 0.2$, $\gamma = 0.0$. For comparison, the position estimate of the target, T, and the position of the query origin, ?, are fixed in all examples (adapted from [43]).

Chapter 5 Sensor Tasking and Control

In all three cases shown in Figure 5.17, the estimated target position and residual uncertainty are the same. Variations in shape and offset of the objective function are caused

by variations of the trade-off parameter γ . In order to visualize how the query is routed toward the maximum of the objective function by local decisions, both the estimated position $\hat{\mathbf{x}}$ as well as its uncertainty are left unaltered during the routing. It is important to note that incremental belief update during the routing by in-network processing would dynamically change both the shape and the offset of the objective function according to the updated values of the estimated position $\hat{\mathbf{x}}$ and its uncertainty at every node along the routing path. As the updated values of $\hat{\mathbf{x}}$ and are passed on to the next node, all routing decisions are still made locally. Hence, the plotted objective function represents a snapshot of the objective function that an active routing node locally evaluates at a given time step, as opposed to the overlaid routing path which illustrates the temporal evolution of the multihop routing.

The small circles surrounding the dots along the routing path illustrate the subset of sensors the routing sensors (on the path) consider during sensor selection. Among these sensors, the ones that locally maximize the objective function have been selected as the successor routing nodes. The fraction of selected nodes among all nodes indicates the energy saved by using the greedy routing, as opposed to the total energy cost of flooding the network. The routing in Figure 5.17 can be terminated after reaching a spatial region where the residual uncertainty is below a preset threshold or the routing has reached a preset timeout that is passed along with the query.

5.4.2 Multistep Information-Directed Routing

The sensor selection in the previous routing problem is *greedy*, always selecting the best sensor given the current belief $p(\mathbf{x} | \{\mathbf{z}_i\}_{i \in U})$, and may get stuck at local maxima caused, for example, by network holes from the depletion of sensor nodes. Figure 5.18(a) provides a simple example. Here we use the inverse of Euclidean distance between a sensor and the target to measure the sensor's information

contribution (assuming these information values are given by an “oracle”). The problem with greedy search exists regardless of the choice of information measure. Consider the case that the target moves from X to Y along a straight line [see Figure 5.18(a)]. Assume nodes A and B are equidistant from the target at any time. At time

$t = 0$, suppose node A is the leader and can relay its target information to its neighbor B or C . If the selection criterion prefers a different node each time to increase diversity, then node B is chosen as the next leader. By the same criteria, B then relays back to A . The hand-offs continue back and forth between A and B , while the target moves away. The path never gets to nodes E , F , or G , who may become more informative as the target moves closer to Y . The culprit in this case is the “sensor hole” the target went through. The greedy algorithm fails due to its lack of knowledge beyond the immediate neighborhood. Recently, local routing algorithms such as GPSR [112] have been developed to traverse perimeters of network holes. However, they do not apply here since the routing destination is not always known a priori in our problem, and it is often impossible to tell if the routing is stuck at a local optimum without knowledge about the destination (e.g., compare the scenario in Figure 5.18(b) with that in Figure 5.18(a)).

Y

F

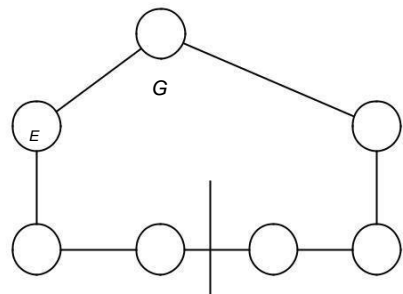
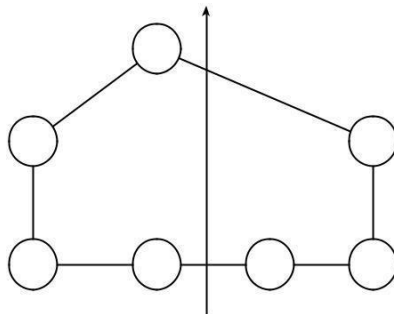
E

F

A

CABDCABD

G



X ∇_x

(a) (b)

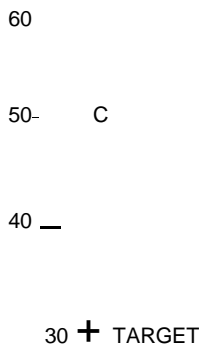
ANNAI WOMEN'S COLLEGE

Figure 5.18 Routing in the presence of sensor holes. A through G are sensor nodes. All edges have unit communication cost. The dashed lines plot target trajectory. In (a), the target is moving from X to Y . In (b), the target is bouncing back and forth between X and Y (adapted from [145]).

er 5 Sensor Tasking and Control

To alleviate the problem of getting trapped at local optima, one may deploy a look-ahead strategy to extend the sensor selection over a finite look-ahead horizon. However, in general the information contribution of each sensor is state-dependent—that is, how much new information a sensor can bring depends on what is already known. This state-dependency property sets the information-directed routing problem apart from traditional routing problems. Standard shortest-path algorithms on graphs such as Dijkstra or Bellman-Ford are no longer applicable. Instead, the path-finding algorithm has to search through many possible paths, leading to combinatorial explosion.

To illustrate the state dependency in information aggregation, consider a simple sensor network example consisting of four sensors, A , B , C , and D , as shown in Figure 5.19. Belief about the target location is shown using grayscale grids (Figure 5.20). A brighter grid means that the target is more likely to be at the grid location.



20

D ○

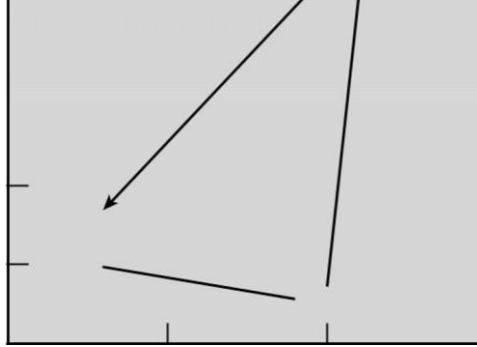
A ○

▶ ○ B

0 0

40

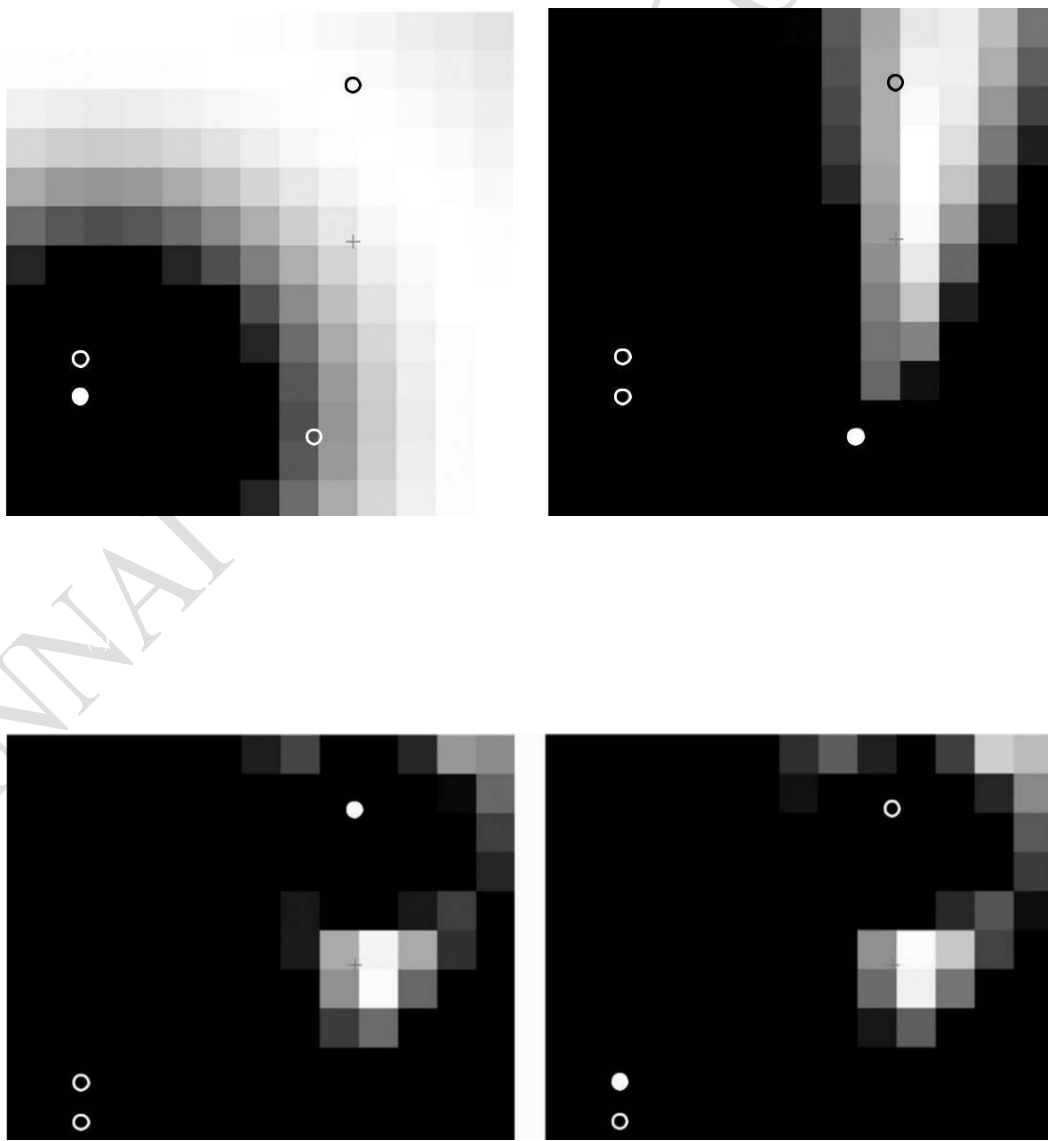
60



ANNAL WOMEN'S COLLEGE

Figure 5.19 A sample sensor network layout: Sensors are marked by circles, with labels *A*, *B*, *C*, and *D*. Arrows represent the order in which sensor data are to be combined. The target is marked by + (adapted from [145]).

We assume a very weak initial belief, uniform over the entire sensor field, knowing only that the target is somewhere in the region. Figure 5.20(a)–(d) shows how information about the target location is updated as sensor data is combined in the order of $A \rightarrow B \rightarrow C \rightarrow D$.



+ +

Figure 5.20 Progressive update of target position, as sensor data is aggregated along the path $ABCD$. Figures (a)–(d) plot the resulting belief after each update.

Table 5.1 Information aggregation in the sensor network pictured in Figure 5.19.

Order of traverse	Information*	MSE
Sensor A	0.67	11.15
Sensor B	1.33	10.02
Sensor C	1.01	9.00
Sensor D	0.07	8.54

*Information is measured using mutual information defined in (5.4).

At each step, the active sensor node, marked as a solid white dot, applies its measurement to update the belief. The localization accuracy is improved over time: the belief becomes more compact and its centroid moves closer to the true target location.

Table 5.1 lists the information contribution for each sensor, as the path $A \rightarrow B \rightarrow C \rightarrow D$ is traversed. The error in localization, measured as mean-squared error (MSE), generally decreases as more sensor measurements are incorporated. Note that sensors A and D are physically near each other, and their contributions toward the target localization should be similar. Despite such similarity, the information values differ significantly (0.67 for A and 0.07 for D). Visually, as can be observed from Figure 5.20, sensor A brings a significant change to the initial uniform belief. In contrast, sensor D hardly causes any change. The reason for the difference is that A applies to a uniform belief state, while D applies to a compact belief, as shown in Figure 5.20(c).

State dependency is an important property of sensor data aggregation, regardless of specific choices of information metrics. Sensor measurements are often correlated. Hence a sensor's measurement is not entirely new; it could be merely repeating what its neighbors have already reported. In our current example, sensor D is highly redundant with sensor A . Such redundancy shows up in the belief state and thus should be discounted. Because of this, the search cost function [say, defined as the path cost minus the information gain, similar to that in (5.1)] is not necessarily additive along a path.

To mitigate the combinatorial explosion problem, two strategies may be useful. We can restrict the search for optimal paths to a

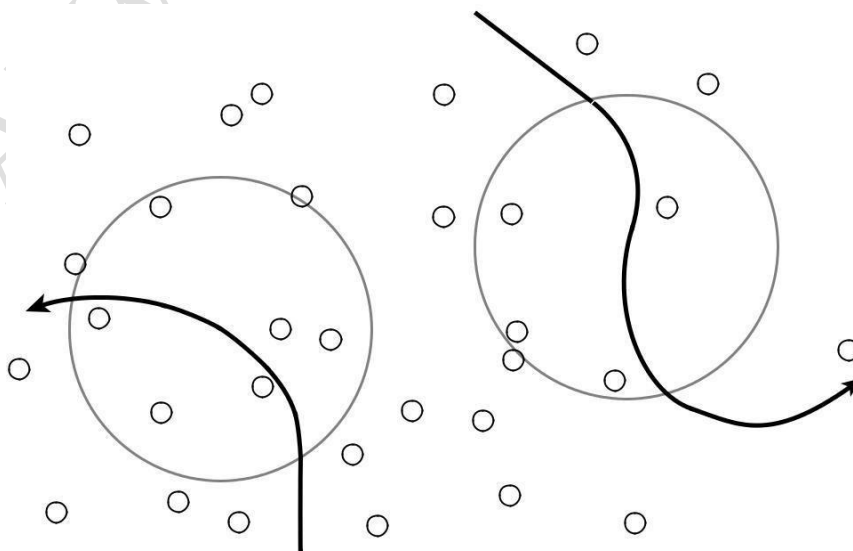
5.4 Joint Routing and Information Aggregation 179

small region of the sensor network. Or we can apply heuristics to approximate the costs so that they can be treated as additive. The first technique we describe searches for a shortest path among the family of paths with fewer than M hops that produce maximum information aggregation. The look-ahead horizon M should be large enough and comparable to the diameter of sensor holes, yet not so large as to make the computational cost prohibitive. The information about network inhomogeneity may be discovered periodically and cached at each node (see, e.g., [102]). Such information will be helpful in selecting the value for M during the routing-path planning.

For scenarios such as the one in Figure 5.16(b), the goal is to route a query from the query proxy to the exit point and accumulate as much information as possible along the way, so that one can extract a good estimate about the target state at the exit node and yet keep the total communication cost close to some prespecified amount. When it is possible to estimate the cost to go, the A^* heuristic search method may be used [120]. The basic A^* is a best-first search, where the merit of a node is assessed as the sum of the actual cost g paid to reach it from the query proxy, and the estimated cost h to pay to get to the exit node (the “cost to go”). For real-time path-finding, we use a variation of the A^* method, namely, the real-time A^* (RTA *) search.¹ It restricts search to a small local region and makes real-time moves before the entire path is planned. Since only local information is used in the RTA * search, it can be implemented in a distributed fashion. Details of the above multistep search algorithms can be found in reference [145].

5.4.3 Sensor Group Management

In the scenarios we have considered so far (Figure 5.16), the physical phenomenon of interest is assumed to be stationary. In many applications, the physical phenomenon may be mobile, requiring the network to migrate the information according to the motion of the



is closely related to the LRTA* algorithm of Section 3.4.4

⊕ Target B

Target A

⊕

Region B

Region A

Figure 5.21 Geographically based collaborative groups. The small circles are sensor nodes. The nodes inside a specified geographical region (e.g., region A or B) form a collaborative group (adapted from [142]).

physical phenomenon for communication efficiency and scalability reasons.

In practical applications, the effect of a physical phenomena usually attenuates with distance, thus limiting the propagation of physical signals to geographical regions around the physical phenomenon. This gives rise to the idea of geographically based collaborative processing groups. In the target tracking problem, for example, one may organize the sensor network into geographical regions, as illustrated in Figure 5.21. Sensors in the region around target *A* are responsible for tracking *A*, those in the region around *B* for tracking *B*. Partitioning the network into local regions assigns network resources according to the potential contributions of individual sensors.

Furthermore, the physical phenomena being sensed change over time. This implies that the collaborative groups also need to be dynamic. As the target moves, the local region must move with it. Sensor nodes that were previously outside the group may join the

5.4 Joint Routing and Information Aggregation **181**

group, and current members may drop out. This requires some method for managing the group membership dynamically.

Geographically based group initiation and management have to be achieved by a lightweight protocol distributed on all sensor nodes. The protocol needs to be powerful enough to handle complex situations, such as those where data from multiple leaders are contending for processing resources, and be robust enough to tolerate poor communication qualities, such as out-of-order delivery and lost or delayed packets. In addition, the propagation region of group management messages should be restrained to only the relevant nodes without flooding the entire network. This is not trivial, considering that the group membership is dynamic as the targets move and that the network is formed in an ad hoc way such that no nodes have the knowledge of the global network topology. The difficulties may be tackled via two techniques: (1) a leader-based tracking algorithm where at any time each group has a unique leader who knows the geographical region of the collaboration; and (2) recent advances in geographical routing (Section 3.4.4) that do not require the leader to know the exact members of its group.

example: Information Migration in Tracking a Moving Target

How can the information utility measures be applied to a tracking problem such as the one described in Section 2.1? Assume a leader node (the solid dot in Figure 5.22) carries the current belief state. The leader chooses a sensor with good information in its neighborhood according to the information measure and then hands off the current belief to the chosen sensor (the new leader). As discussed earlier, the information-based approach to sensor querying and data routing selectively invokes sensors to minimize the number of sensing actions needed for a given accuracy and, hence, latency and energy usage.

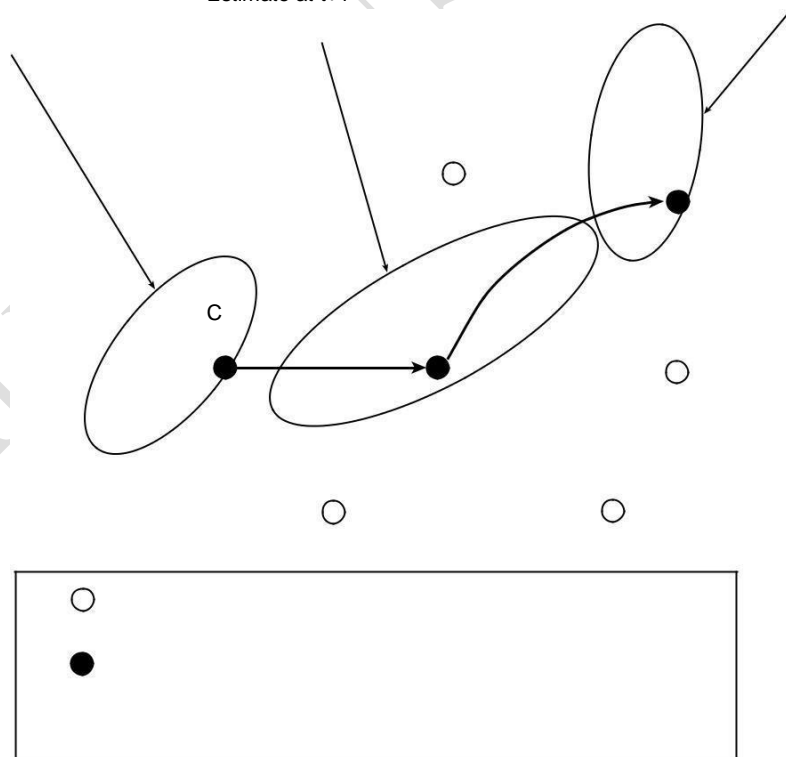
To estimate the position of the target, a leader node updates the belief state information received from the previous leader with the current measurement information, using, for example, the sequential Bayesian method introduced in Section 2.2.3. For a moving target, a model on the target dynamics can be used to predict the

apter 5 Sensor Tasking and Control

estimate at $t+2$

Estimate at $t+1$

Estimate at t



A

B

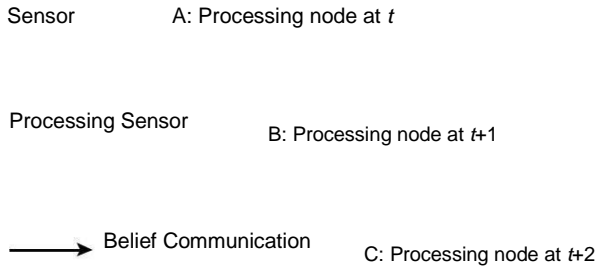


Figure 5.22 Information migration in tracking a moving target. As the target moves through the field of sensors, a subset of sensors are activated to carry the belief state. Each new sensor may be selected according to an information utility measure on the expected contribution of that sensor conditioned on the predicted location of the target (adapted from [41]).

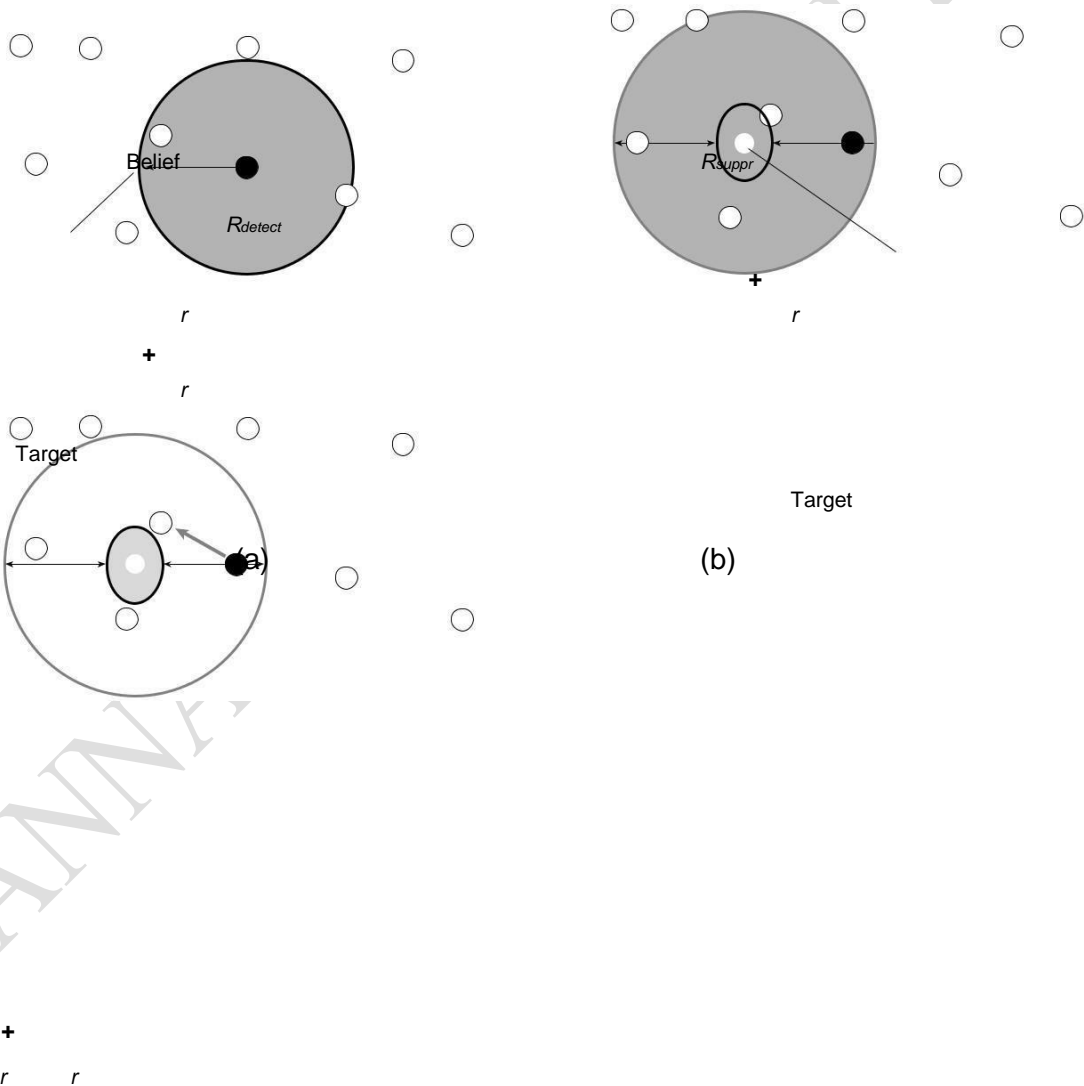
position of the target at the next time step. This predicted target position and the associated uncertainty can be used to dynamically aim the information-directed query at future positions to optimally track the target.

Distributed Group Management

A collaborative group is a set of sensor nodes responsible for the creation and maintenance of a target's belief state over time, which we call a *track*. Effectively, these are sensors whose coverage overlaps

with the state estimate of the track. When the target enters the sensor field or emits a signal for the first time, it is detected by a set of sensor nodes. Each individual sensor performs a local detection using a likelihood ratio test. Nodes with detections form a collaborative group and select a single leader—for instance, based on time of detection as discussed in Section 5.3.3. While we discuss the single-leader approach, it is also possible that a small number of nodes are elected to share the leadership. Figure 5.23 shows how the leader node maintains and migrates the collaborative processing group.

After the leader is elected, it initializes a belief state $p(\mathbf{x}|\mathbf{z})$ as a uniform disk R_{detect} centered at its own location [Figure 5.23(a)].



(c)

Figure 5.23 Distributed group maintenance: (a) An elected leader initializes a uniform belief over

a region R_{detect} . (b) The leader estimates the target position and sends a suppression message to a region R_{suppr} . Nodes not receiving suppression time out to detection

mode. (c) A new leader is selected using a sensor selection criterion. The current belief state is handed off to the new leader.

Chapter 5 Sensor Tasking and Control

The disk contains the true target location with high probability.

This belief provides a starting point for the subsequent tracking.

As the target moves, the sensors that did not previously detect may begin detecting. These sensors are potential sources of contention. The leader uses the uncertainty in track position estimate and maximum detection range to calculate a suppression region and informs all group members in the suppression region to stop detection [Figure 5.23(b)]. This reduces energy consumption of the other nodes and avoids further track initiation. The assumption is that there is only one target in the neighborhood.

Sensors are selected to acquire new measurements, using, for example, the sensor selection algorithm discussed earlier [Figure 5.23(c)]. As the belief state is refined by successive measurements, the group membership needs to be updated. This is accomplished by updating the suppression region using suppression and unsuppression messages to designated regions.

When the targets are far apart, their tracks are handled by multiple collaborative groups working in parallel. When targets cross, the position uncertainty regions for their tracks overlap and the collaborative groups for these tracks are no longer distinct. This can be detected when a leader node receives a suppression message from a node with a different (track) ID from its own. When two groups collide, the sensor measurements in the overlapping region can now be associated with either one of the two tracks.

Data association algorithms such as optimal assignment or multiple hypothesis processing can be used to resolve this ambiguity. For example, a simple track-merging approach is to keep the older track and drop the younger track. The two collaborative groups then merge into a single group. This approach works well if the two tracks were initiated from a single target. When the two tracks result from two targets, the merging operation will temporarily track the two targets as one. When they separate again, a new track corresponding to one of the targets will be reinitiated; however, the identities of the targets will be lost. Using an identity management algorithm,

5.4 Joint Routing and Information Aggregation **185**

the ambiguities in the target identities after crossing tracks can be resolved using additional local evidence of the track identity and then propagate the information to other relevant tracks. Details of managing groups for multiple targets and their identities can be found in references [142, 210].

5.4.4 Case Study: Sensing Global Phenomena

We have been primarily concerned with sensing point targets so far. In some situations, we might encounter the problem of sensing a global phenomenon using local measurements only. In Section 5.3.4, we briefly described the problem of sensing a global relation among a set of objects. Another example of sensing global phenomena is determining and tracking the boundary of a large object moving over a sensor field, where each sensor only “sees” a portion of the boundary. One such application is tracking a moving chemical plume using airborne and ground-based chemical sensors.

How does the sensor tasking for these problems differ from what we have considered in sensing point targets? A primary challenge is to relate a local sensing action to the utility of determining the global property of the object(s) of interest. To address this challenge, we need to convert the global estimation and tracking problem into a local analysis, using, for example, the so-called *primal-dual transformation* [51, 140]. Just as a Fourier transform maps a global property of a signal such as periodicity in the time domain to a local feature in the frequency domain, the primal-dual transformation maps a line in the primal space into a point in the dual space, and vice versa (Figure 5.24). Using the primal-dual transformation, the shape of a target, when approximated as a polygonal object, can be tracked as a set of points in the dual space.

A useful consequence of this mapping is its use in tasking sensors to sense a global phenomenon such as the boundary of a moving half-plane shadow. As we noted earlier, a wireless sensor network is severely constrained by the on-board battery power. If a sensor only wakes up, senses, and communicates when it expects an event of interest, the power consumption of the network could

Chapter 5 Sensor Tasking and Control

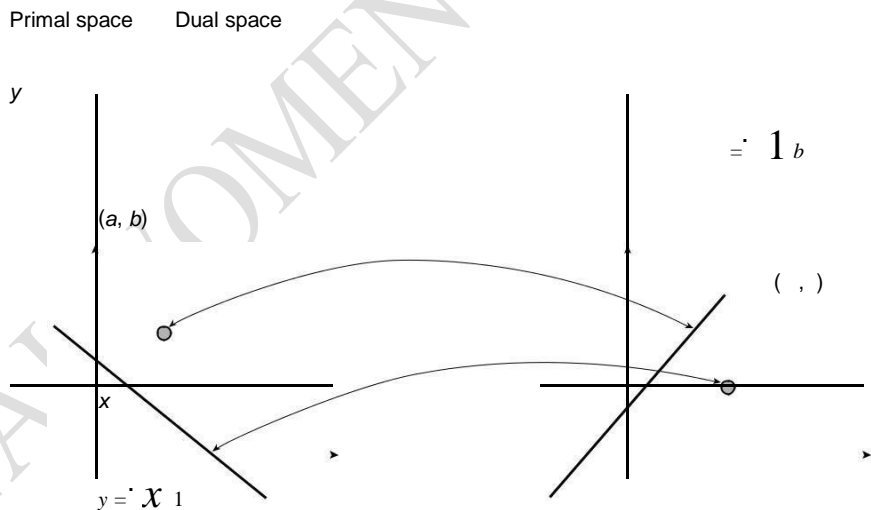


Figure 5.24 Primal-dual transformation. It is a one-one mapping where a point maps to a line and a line maps to a point (adapted from [140]).

be dramatically reduced. For the boundary tracking problem, the prediction of when a sensor needs to participate in a collaborative processing task can be made in the dual-space

representation, where a boundary line L in the primal space is represented as a point l in the dual space, and sensors (points in the primal space) are represented as lines (Figure 5.25). Those lines that form a cell containing the point l correspond to sensors that are potentially relevant for the next sensing task. As the half-plane shadow moves in the physical space (i.e., primal space), the corresponding point in the dual space moves from cell to cell. When the point crosses the cell boundary,

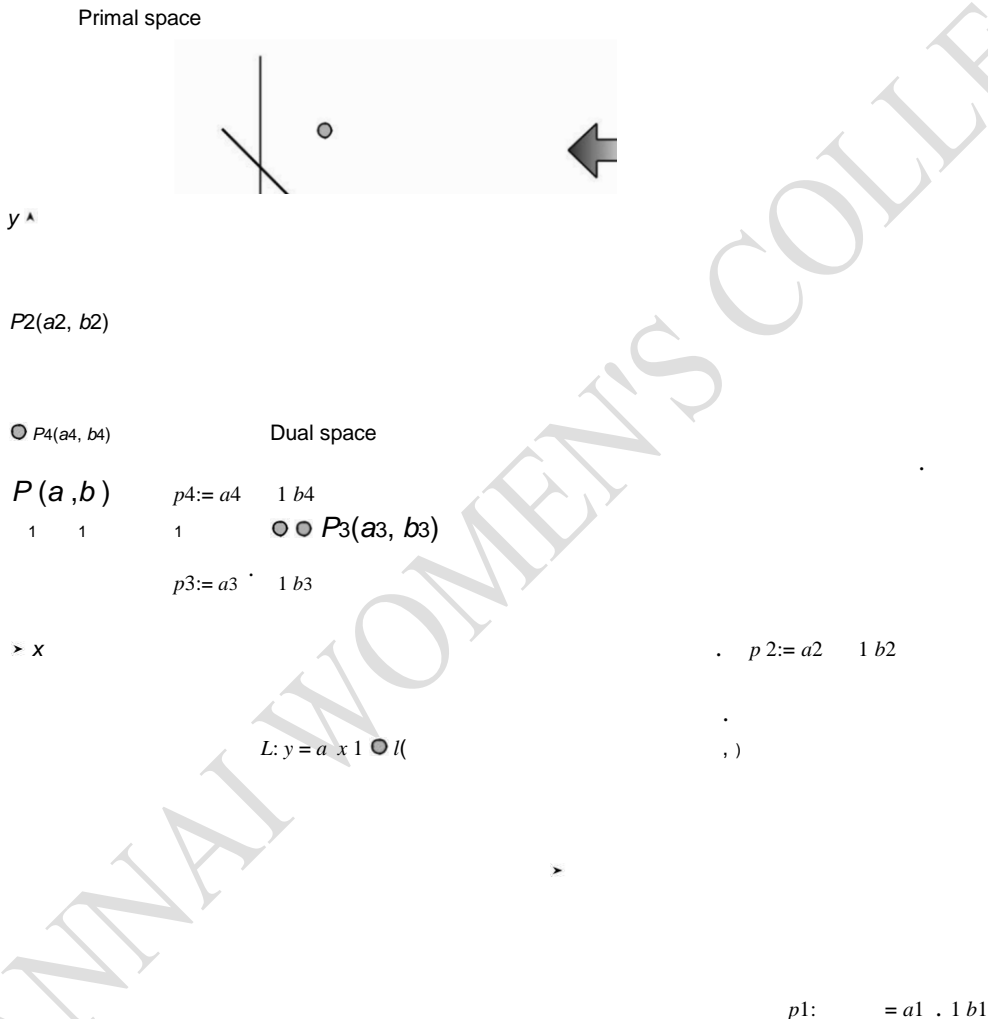


Figure 5.25 The prediction of which sensors will be relevant for sensing the target boundary L in the primal space is equivalent to the determination of lines that form the cell containing the corresponding l in the dual space arrangement (adapted from [140]).

a set of new sensors may become relevant to sensing the half-plane boundary and can be read off from the lines enclosing the current position in the dual space. The number of lines bounding a cell is 4 on the average. Thus the number of sensors that need be active at once is very small, no matter how many sensors are present in the field. This idea has been implemented and tested on a testbed of Berkeley wireless sensor motes (see reference [140]). An open problem that remains to be addressed is an effective decentralization of the computation in the dual-space containment test.

5.5 Summary

We have developed a number of important ideas for efficiently allocating the sensing, processing, and communication resources of a sensor network to monitoring and other application tasks. We have introduced the models of information utility and costs, as a basis for decentralized coordination and optimization *within* the network. The idea of information-driven sensor tasking, and its realization in IDSQ, is to base the sensor selection on the potential contribution of a sensor to the current estimation task while using a moderate amount of resources. Applying the idea to sensing stationary or moving physical phenomena, we developed a number of protocols, including the leader-based and moving center of aggregation. Moving beyond local greedy sensor selection, we introduced an information-driven routing to jointly optimize for routing and information aggregation, using a multistep look-ahead search. We also touched on the important topic of creating and managing collaborative processing sensor groups, which are common to a number of monitoring applications.

A number of key themes emerge from these discussions:

Central to these ideas is the notion of information utility, and the associated costs of acquiring the information. In the resource-limited sensor networks, the appropriate balance between the information and the costs is of paramount concern, since

Chapter 5 Sensor Tasking and Control

unnecessary data collection or communication consumes precious bandwidth and energy and overload human attention.

The information utility measures can take on many different forms, depending on the application requirement and context. Aside from information-theoretic considerations, we must carefully evaluate the computational complexity of applying the utility measures to sensor tasking, as inappropriate uses of information utility may consume intolerable amounts of resources and thereby diminish the benefit.

As a sensor network's primary function is to collect information from a physical environment, we must rethink the role of routing in this context. As is becoming clear in the examples we

examined, routing in a sensor network often does not just perform a pure information transport function. It must be co-optimized with the information aggregation or dissemination.

A collaborative processing group is an important abstraction of physical sensors, since individual sensors are ephemeral and hence less important, and sensors collectively support a set of tasks. The challenge is to efficiently create, maintain, and migrate groups as tasks and physical environments change. A major benefit of establishing the collaborative group abstraction is in enabling the programming of sensor networks to move from addressing individual nodes to addressing collectives, a topic we discuss again in the context of platform issues and programming models in Chapter 7.

Sensor Network Platforms and Tools

In previous chapters, we discussed various aspects of sensor networks, including sensing and estimation, networking, infrastructure services, sensor tasking, and data storage and query. A real-world sensor network application most likely has to incorporate all these elements, subject to energy, bandwidth, computation, storage, and real-time constraints. This makes sensor network application development quite different from traditional distributed system development or database programming. With ad hoc deployment and frequently changing network topology, a sensor network application can hardly assume an always-on infrastructure that provides reliable services such as optimal routing, global directories, or service discovery.

There are two types of programming for sensor networks, those carried out by end users and those performed by application developers. An end user may view a sensor network as a pool of data and *interact* with the network via queries. Just as with query languages for database systems like SQL, a good sensor network programming language should be expressive enough to encode application logic at a high level of abstraction, and at the same time be structured enough to allow efficient execution on the distributed platform. Examples of sensor database query interfaces are described in Chapter 6. Ideally, the end users should be shielded away from details of how sensors are organized and how nodes communicate.

On the other hand, an application developer must provide end users of a sensor network with the capabilities of data acquisition, processing, and storage. Unlike general distributed or database systems, collaborative signal and information processing (CSIP)

ANNAL WOMEN'S COLLEGE

software comprises reactive, concurrent, distributed programs running on ad hoc, resourceconstrained, unreliable computation and communication platforms. Developers at this level have to deal with all kinds of uncertainty in the real world. For example, signals are noisy, events can happen at the same time, communication and computation take time, communications may be unreliable, battery life is limited, and so on. Moreover, because of the amount of domain knowledge required, application developers are typically signal and information processing specialists, rather than operating systems and networking experts. How to provide appropriate programming abstractions to these application writers is a key challenge for sensor network software development. In this chapter, we focus on software design issues to support this type of programming.

To make our discussion of these software issues concrete, we first give an overview of a few representative sensor node hardware platforms (Section 7.1). In Section 7.2, we present the challenges of sensor network programming due to the massively concurrent interaction with the physical world. Section 7.3 describes TinyOS for Berkeley nodes and two types of node-centric programming interfaces: an imperative language, nesC, and a dataflow-style language, TinyGALS. Node-centric designs are typically supported by node-level simulators such as ns-2 and TOSSIM, as described in Section 7.4. State-centric programming is a step toward programming beyond individual nodes. It gives programmers platform support for thinking in high-level abstractions, such as the state of the phenomena of interest over space and time. An example of state-centric platforms is given in Section 7.5.

7.1 Sensor Node Hardware

Sensor node hardware can be grouped into three categories, each of which entails a different set of trade-offs in the design choices.

Augmented general-purpose computers: Examples include low-power PCs, embedded PCs (e.g., PC104), custom-designed PCs

(e.g., Sensoria WINS NG nodes),¹ and various personal digital assistants (PDA). These nodes typically run off-the-shelf operating systems such as Win CE, Linux, or real-time operating systems and use standard wireless communication protocols such as Bluetooth or IEEE 802.11. Because of their relatively higher processing capability, they can accommodate a wide variety of sensors, ranging from simple microphones to more sophisticated video cameras.

Compared with dedicated sensor nodes, PC-like platforms are more power hungry. However, when power is not an issue, these platforms have the advantage that they can leverage the availability of fully supported networking protocols, popular programming languages, middleware, and other off-the-shelf software.

Dedicated embedded sensor nodes: Examples include the Berkeley mote family [98], the UCLA Medusa family [202], Ember nodes,² and MIT μ AMP [32]. These platforms typically use commercial off-the-shelf (COTS) chip sets with emphasis on small form factor, low power processing and communication, and simple sensor interfaces. Because of their COTS CPU, these platforms typically support at least one programming language, such as C. However, in order to keep the program footprint small to accommodate their small memory size, programmers of these platforms are given full access to hardware but barely any operating system support. A classical example is the TinyOS platform and its companion programming language, nesC. We will discuss these platforms in Sections 7.3.1 and 7.3.2.

System-on-chip (SoC) nodes: Examples of SoC hardware include smart dust [109], the BWRC picoradio node [187], and the PASTA node.³ Designers of these platforms try to push the hardware limits by fundamentally rethinking the hardware architecture trade-offs for a sensor node at the chip design level. The goal is to find new ways of integrating CMOS, MEMS, and RF technologies

See <http://www.sensoria.com/> and <http://www.janet.ucla.edu/WINS/>, and [158].

² See <http://www.ember.com>.

³ See <http://pasta.east.isi.edu>.


Chapter 7 Sensor Network Platforms and Tools

to build extremely low power and small footprint sensor nodes that still provide certain sensing, computation, and communication capabilities. Since most of these platforms are currently in the research pipeline with no predefined instruction set, there is no software platform support available.

Among these hardware platforms, the Berkeley motes, due to their small form factor, open source software development, and commercial availability, have gained wide popularity in the sensor network research community. In the following section, we give an overview of the Berkeley MICA mote.

7.1.1 Berkeley Mote

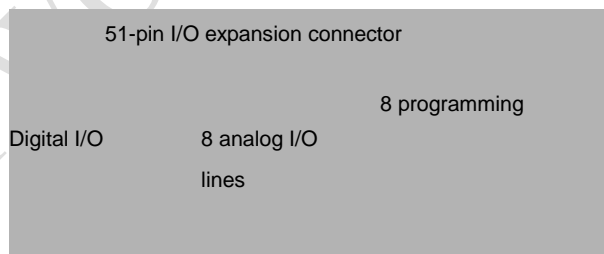
The Berkeley motes are a family of embedded sensor nodes sharing roughly the same architecture. Figure 7.1 shows a comparison of a subset of mote types.

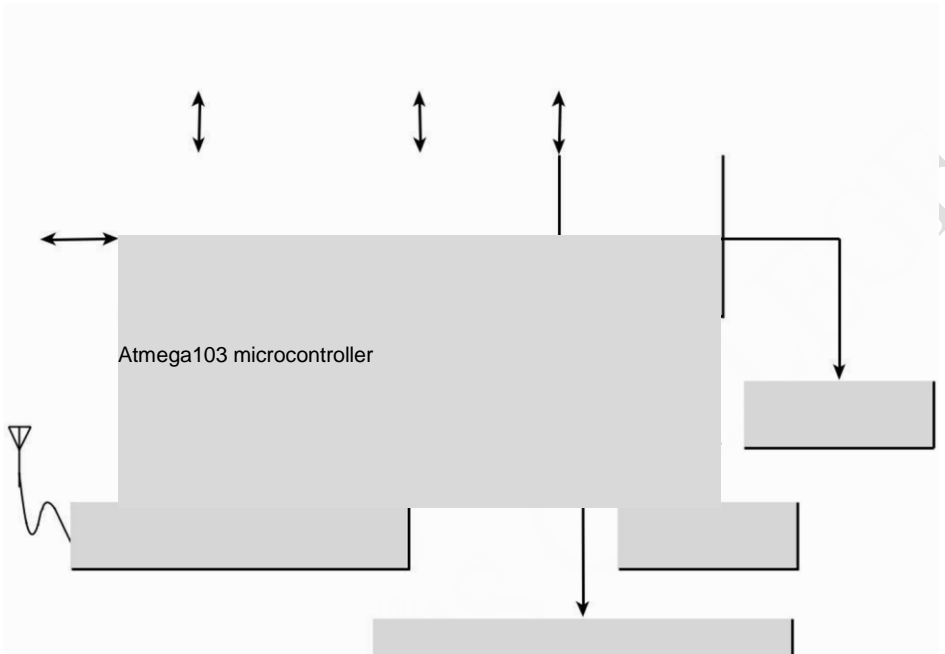
Mote type		WeC	Rene	Rene2	Mica
ample Ex picture					
MCU	Chip	AT90LS8535	ATmega163L	ATmega103L	
	Type	4 MHz, 8 bit	4 MHz, 8 bit	4 MHz, 8 bit	
	Program memory (KB)	8	16	128	
	RAM (KB)	0.5	1	4	
External nonvolatile storage	Chip	24 LC 256			
	Connection type	I ² C			
	Size (KB)	32			
Default power source	Type	Coin cell			
	Typical capacity (mAh)	575		2850	
	Chip	TR 1000			

RF	Radio frequency	868/9 16MHz	
	Raw speed (kbps)	10	40
	Modulation type	On/Off key	Amplitude Shift key

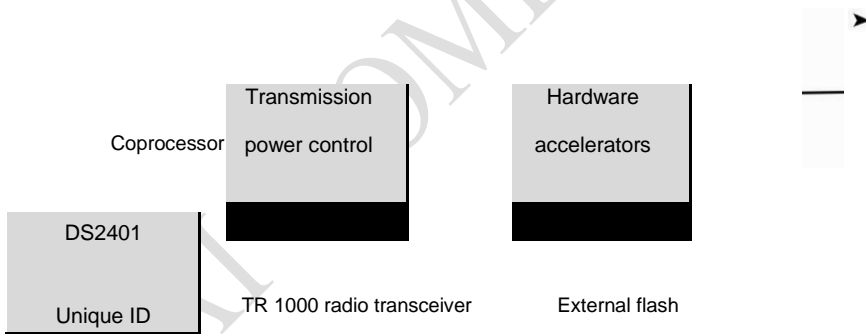
Figure 7.1 A comparison of Berkeley motes.

Let us take the MICA mote as an example. The MICA motes have a two-CPU design, as shown in Figure 7.2. The main microcontroller (MCU), an Atmel ATmega103L, takes care of regular processing. A separate and much less capable coprocessor is only active when the MCU is being reprogrammed. The ATmega103L MCU has integrated 512 KB flash memory and 4 KB of data memory. Given these small memory sizes, writing software for motes is challenging. Ideally, pro-grammers should be relieved from optimizing code at assembly level to keep code footprint small. However, high-level support and soft-ware services are not free. Being able to mix and match only necessary software components to support a particular application is essential to achieving a small footprint. A detailed discussion of the software architecture for motes is given in Section 7.3.1.





Wireless Sensor Networks S.Leelavathi



Power regulation MAX1678 (3V)

Figure 7.2 MICA mote architecture.

Chapter 7 Sensor Network Platforms and Tools

In addition to the memory inside the MCU, a MICA mote also has a separate 512 KB flash memory unit that can hold data. Since the connection between the MCU and this external memory is via a low-speed serial peripheral interface (SPI) protocol, the external memory is more suited for storing data for later batch processing than for storing programs. The RF communication on MICA motes uses the TR1000 chip set (from RF Monolithics, Inc.) operating at 916 MHz band. With hardware accelerators, it can achieve a maximum of 50 kbps raw data rate. MICA motes implement a 40 kbps transmission rate. The transmission power can be digitally adjusted by software through a potentiometer (Maxim DS1804). The maximum transmission range is about 300 feet in open space.

Like other types of motes in the family, MICA motes support a 51 pin I/O extension connector. Sensors, actuators, serial I/O boards, or parallel I/O boards can be connected via the connector. A sensor/ actuator board can host a temperature sensor, a light sensor, an accelerometer, a magnetometer, a microphone, and a beeper. The serial I/O (UART) connection allows the mote to communicate with a PC in real time. The parallel connection is primarily for downloading programs to the mote.

It is interesting to look at the energy consumption of various components on a MICA mote. As shown in Figure 7.3, a radio

<i>Component</i>	<i>Rate</i>	<i>Startup time</i>	<i>Current consumption</i>
MCU active	4 MHz	N/A	5.5 mA
MCU idle	4 MHz	1 μ s	1.6 mA

162

Wireless Sensor Networks S.Leelavathi

MCU suspend	32 kHz	4 ms	<20 μ A
Radio transmit	40 kHz	30 ms	12 mA
Radio receive	40 kHz	30 ms	1.8 mA
Photoresistor	2000 Hz	10 ms	1.235 mA
Accelerometer	100 Hz	10 ms	5 mA/axis
Temperature	2 Hz	500 ms	0.150 mA

Figure 7.3

Power consumption of MICA motes.


transmission bears the maximum power consumption. However, each radio packet (e.g., 30 bytes) only takes 4 ms to send, while listening to incoming packets turns the radio receiver on all the time. The energy that can send one packet only supports the radio receiver for about 27 ms. Another observation is that there are huge differences among the power consumption levels in the active mode, the idle mode, and the suspend mode of the MCU. It is thus worthwhile from an energy-saving point of view to suspend the MCU and the RF receiver as long as possible.

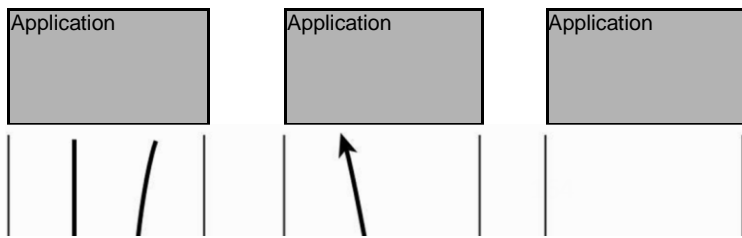
7.2 Sensor Network Programming Challenges

Traditional programming technologies rely on operating systems to provide abstraction for processing, I/O, networking, and user interaction hardware, as illustrated in Figure 7.4. When applying such a

React to all events/messages

(typically using an FSM)





Wireless Sensor Networks S.Leelavathi

- Message passing
- Handshaking
- Locks and monitors
- Interrupt services
- Polling sensors

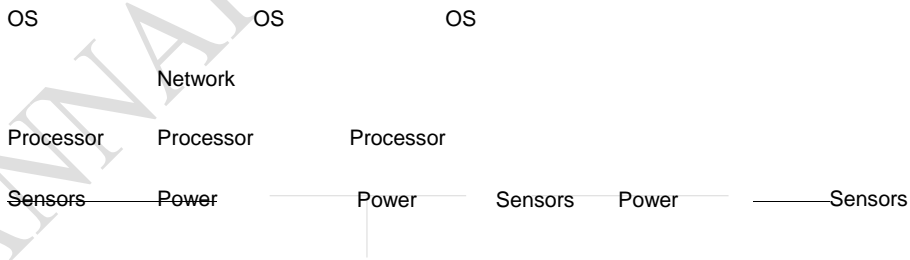


Figure 7.4 Traditional embedded system programming interface.

ANNAL WOMEN'S COLLEGE

Chapter 7 Sensor Network Platforms and Tools

model to programming networked embedded systems, such as sensor networks, the application programmers need to explicitly deal with message passing, event synchronization, interrupt handling, and sensor reading. As a result, an application is typically implemented as a finite state machine (FSM) that covers all extreme cases: unreliable communication channels, long delays, irregular arrival of messages, simultaneous events, and so on. In a target tracking application implemented on a Linux operating system and with directed diffusion routing, roughly 40 percent of the code implements the FSM and the glue logic of interfacing computation and communication [142].

For resource-constrained embedded systems with real-time requirements, several mechanisms are used in embedded operating systems to reduce code size, improve response time, and reduce energy consumption. Microkernel technologies [211] modularize the operating system so that only the necessary parts are deployed with the application. Realtime scheduling [27] allocates resources to more urgent tasks so that they can be finished early. Event-driven execution allows the system to fall into low-power sleep mode when no interesting events need to be processed. At the extreme, embedded operating systems tend to expose more hardware controls to the programmers, who now have to directly face device drivers and scheduling algorithms, and optimize code at the assembly level. Although these techniques may work well for small, stand-alone embedded systems, they do not scale up for the programming of sensor networks for two reasons.

Sensor networks are large-scale distributed systems, where global properties are derivable from program execution in a massive number of distributed nodes. Distributed algorithms themselves are hard to implement, especially when infrastructure support is limited due to the ad hoc formation of the system and constrained power, memory, and bandwidth resources.

As sensor nodes deeply embed into the physical world, a sensor network should be able to respond to multiple concurrent stimuli at the speed of changes of the physical phenomena of interest.

In the rest of the chapter, we give several examples of sensor network software design platforms. We discuss them in terms of both *design methodologies* and *design platforms*. A design methodology implies a conceptual model for programmers, with associated techniques for problem decomposition for the software designers. For example, does the programmer think in terms of events, message passing, and synchronization, or does he/she focus more on information architecture and data semantics? A design platform supports a design methodology by providing design-time (precompile time) language constructs and restrictions, and run-time (postcompile time) execution services.

There is no single universal design methodology for all applications. Depending on the specific tasks of a sensor network and the way the sensor nodes are organized, certain methodologies and platforms may be better choices than others. For example, if the network is used for monitoring a small set of phenomena and the sensor nodes are organized in a simple star topology, then a client-server software model would be sufficient. If the network is used for monitoring a large area from a single access point (i.e., the base station), and if user queries can be decoupled into aggregations of sensor readings from a subset of sensor nodes, then a tree structure that is rooted at the base station is a better choice. However, if the phenomena to be monitored are moving targets, as in the target tracking examples discussed in Chapter 2, then neither the simple client-server model nor the tree organization is optimal. More sophisticated design methodologies and platforms are required.

7.3 Node-Level Software Platforms

Most design methodologies for sensor network software are node-centric, where programmers think in terms of how a node should behave in the environment. A node-level platform can be a node-centric operating system, which provides hardware and networking abstractions of a sensor node to programmers, or it can be a language platform, which provides a library of components to programmers.

A typical operating system abstracts the hardware platform by providing a set of services for applications, including file management, memory allocation, task scheduling, peripheral device drivers, and networking. For embedded systems, due to their highly specialized applications and limited resources, their operating systems make different trade-offs when providing these services. For example, if there is no file management requirement, then a file system is obviously not needed. If there is no dynamic memory allocation, then memory management can be simplified. If prioritization among tasks is critical, then a more elaborate priority scheduling mechanism may be added.

TinyOS [98] and TinyGALS [38] are two representative examples of node-level programming tools that we will cover in detail in this section. Other related software platforms include Maté [130], a virtual machine for the Berkeley motes. Observing that

operations such as polling sensors and accessing internal states are common to all sensor network application, Maté defines virtual machine instructions to abstract those operations. When a new hardware platform is introduced with support for the virtual machine, software written in the Maté instruction set does not have to be rewritten.

7.3.1 Operating System: TinyOS

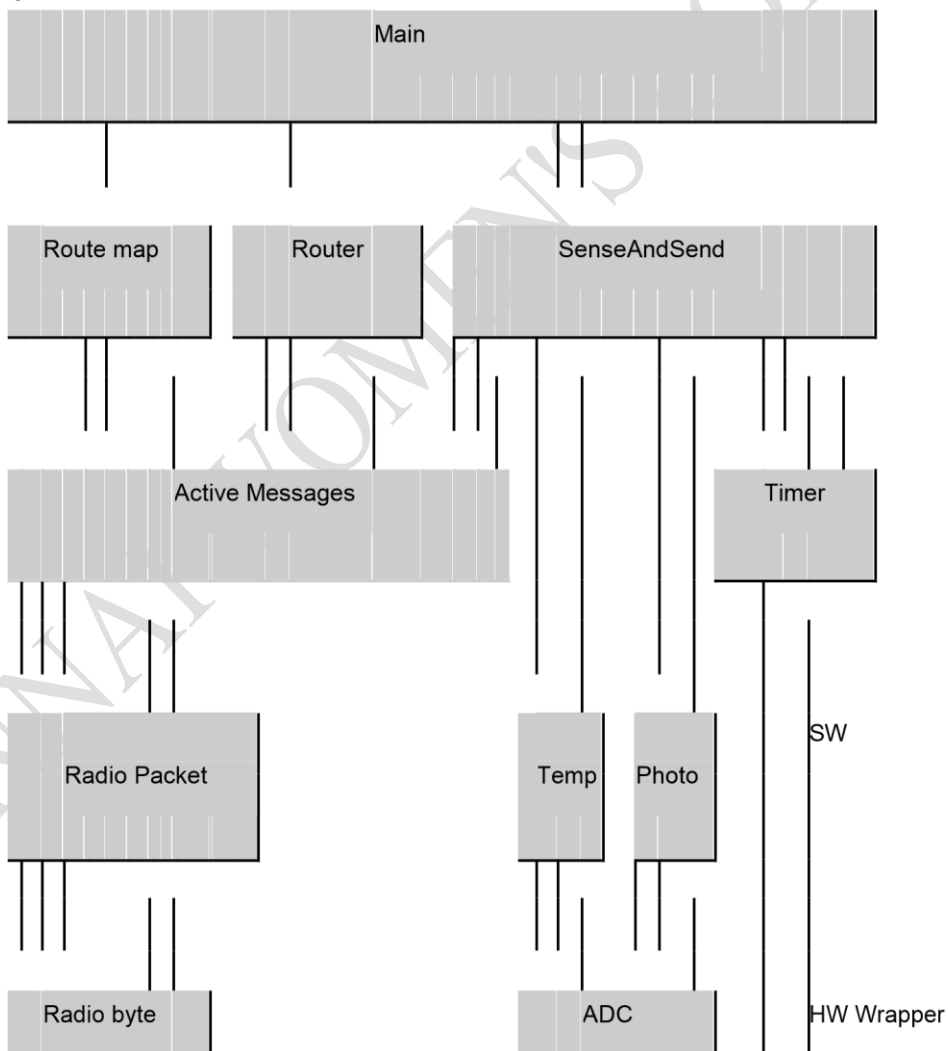
TinyOS aims at supporting sensor network applications on resource-constrained hardware platforms, such as the Berkeley motes.

To ensure that an application code has an extremely small foot-print, TinyOS chooses to have no file system, supports only static memory allocation, implements a simple task model, and provides minimal device and networking abstractions. Furthermore, it takes a language-based application development approach, to be discussed later, so that only the necessary parts of the operating system are compiled with the application. To a certain extent, each TinyOS application is built into the operating system.

Like many operating systems, TinyOS organizes components into layers. Intuitively, the lower a layer is, the “closer” it is to the hardware; the higher a layer is, the “closer” it is to the application.

In addition to the layers, TinyOS has a unique component architecture and provides as a library a set of system software components. A component specification is independent of the component implementation. Although most components encapsulate software functionalities, some are just thin wrappers around hardware. An application, typically developed in the nesC language covered in the next section, *wires* these components together with other application-specific components.

Let us consider a TinyOS application example—FieldMonitor, where all nodes in a sensor field periodically send their temperature and photo sensor readings to a base station via an ad hoc routing mechanism. A diagram of the FieldMonitor application is shown in Figure 7.5, where blocks represent TinyOS components and arrows represent function calls among them. The directions of the arrows are from callers to callees.



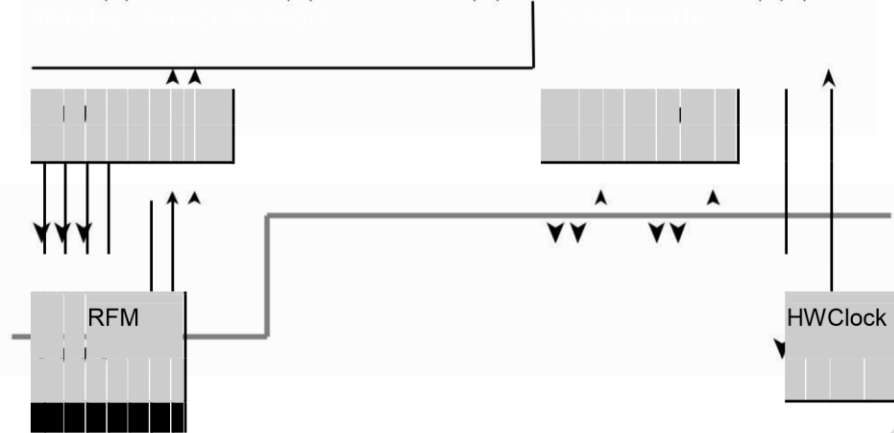


Figure 7.5 The FieldMonitor application for sensing and sending measurements.

ANNAI WOMEN'S COLLEGE

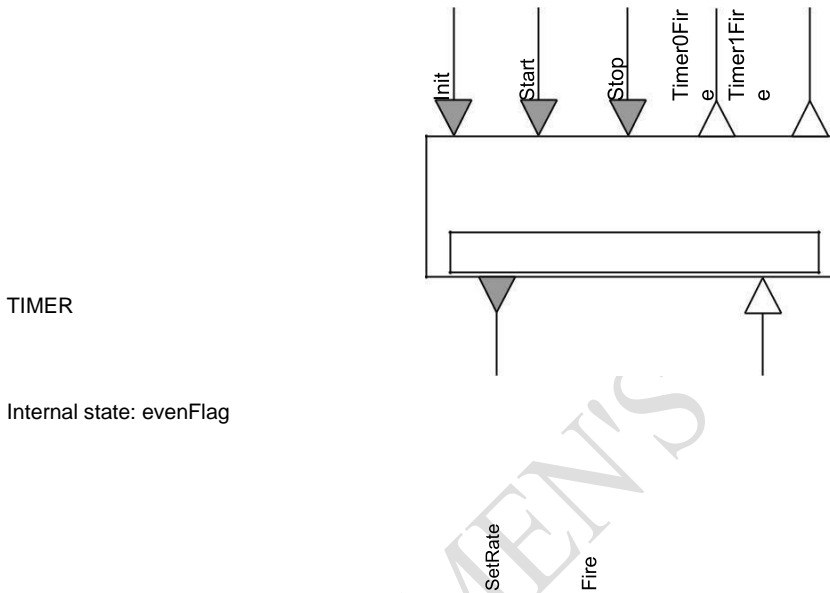


Figure 7.6 The Timer component and its interfaces.

To explain in detail the semantics of TinyOS components, let us first look at the Timer component of the FieldMonitor application, as shown in Figure 7.6. This component is designed to work with a clock, which is a software wrapper around a hardware clock that generates periodic interrupts. The method calls of the Timer component are shown in the figure as the arrowheads. An arrowhead pointing into the component is a method of the component that other components can call. An arrowhead pointing outward is a method that this component requires another layer component to provide. The absolute directions

of the arrows, up or down, illustrate this component's relationship with other layers. For example, the Timer depends on a lower layer HWClock component. The Timer can set the

ANNAL WOMEN'S COLLEGE

rate of the clock, and in response to each clock interrupt it toggles an internal Boolean flag, `evenFlag`, between true (or 1) and false (or 0). If the flag is 0, the Timer produces a `timerOFire` event to trigger other components; otherwise, it produces a `timerIFire` event. The Timer has an `init()` method that initializes its internal flag, and it can be enabled and disabled via the `start` and `stop` calls.

A program executed in TinyOS has two contexts, *tasks* and *events*, which provide two sources of concurrency. Tasks are created (also called *posted*) by components to a task scheduler. The default

implementation of the TinyOS scheduler maintains a task queue and invokes tasks according to the order in which they were posted. Thus tasks are deferred computation mechanisms. Tasks always run to completion without preempting or being preempted by other tasks. Thus tasks are nonpreemptive. The scheduler invokes a new task from the task queue only when the current task has completed. When no tasks are available in the task queue, the scheduler puts the CPU into the sleep mode to save energy.

The ultimate sources of triggered execution are events from hardware: clock, digital inputs, or other kinds of interrupts. The execution of an interrupt handler is called an *event context*. The processing of events also runs to completion, but it preempts tasks and can be pre-empted by other events. Because there is no preemption mechanism among tasks and because events always preempt tasks, programmers are required to chop their code, especially the code in the event contexts, into small execution pieces, so that it will not block other tasks for too long.

notify its caller by a `sendDone()` method call. Only at this time is the AM component ready to accept another packet.

In TinyOS, resource contention is typically handled through explicit rejection of concurrent requests. All split-phase operations return Boolean values indicating whether a request to perform the operation is accepted. In the above example, a call of `send()`, when the AM component is still sending the first packet, will result in an error signaled by the AM component. To avoid such an error, the caller of the AM component typically implements a *pending* lock, to remember not to request further sendings until the `sendDone()` method is called. To avoid loss of packets, a queue should be incorporated by the caller if necessary.

In summary, many design decisions in TinyOS are made to ensure that it is extremely lightweight. Using a component architecture that contains all variables inside the components and disallowing dynamic memory allocation reduces the memory management overhead and makes the data memory usage statically analyzable. The simple concurrency

model allows high concurrency with low thread maintenance overhead. As a consequence, the entire FieldMonitor system shown in Figure 7.5 takes only 3 KB of space for code and 226 bytes for data. However, the advantage of being lightweight is not without cost. Many hardware idiosyncrasies and complexities of concurrency management are left for the application programmers to handle. Several tools have been developed to give programmers language-level support for improving programming productivity and code robustness. We introduce in the next two sections two special-purpose languages for programming sensor network nodes. Although both languages are designed on top of TinyOS, the principles they represent may apply to other platforms.

7.3.2 Imperative Language: nesC

nesC [79] is an extension of C to support and reflect the design of TinyOS v1.0 and above. It provides a set of language constructs and restrictions to implement TinyOS components and applications.

Component Interface

A component in nesC has an interface specification and an implementation. To reflect the layered structure of TinyOS, interfaces of a nesC component are classified as *provides* or *uses* interfaces. A provides interface is a set of method calls exposed to the upper layers, while a uses interface is a set of method calls hiding the lower layer components. Methods in the interfaces can be grouped and named. For example, the interface specification of the Timer component in Figure 7.6 is listed in Figure 7.7. The interface, again, independent of the implementation, is called TimerModule.

Although they have the same method call semantics, nesC distinguishes the *directions* of the interface calls between layers as *event* calls

```
module TimerModule {  
  provides {  
    interface StdControl;  
    interface Timer01;  
  }  
  uses {  
    interface StdControl;  
    interface Timer01;  
  }  
}
```

```

a
c
e
C
l
o
c
k
a
s
C
l
k
;

}

interface StdControl {

command result_t init();

}

interface Timer01 {

command result_t start(char type, uint32_t interval;

command result_t stop();
event result_t timer0Fire();
event result_t timer1Fire();

}

interface Clock {
command result_t setRate(char interval, char scale); event result_t
fire(); }

```

Figure 7.7 The interface definition of the Timer component in nesC.

and *command* calls. An event call is a method call from a lower layer component to a higher layer component, while a command is the opposite. Note that one needs to know both the type of the interface (provides or uses) and the direction of the method call (event or command) to know exactly whether an interface method is implemented by the component or is required by the component.

The separation of interface type definitions from how they are used in the components promotes the reusability of standard interfaces. A component can provide and use the same

interface type, so that it can act as a filter interposed between a client and a service. A component may even use or provide the same interface multiple times. In these cases, the component must give each interface instance a separate name using the `as` notation, as shown in the `Clock` interface in Figure 7.7.

Component Implementation

There are two types of components in `nesC`, depending on how they are implemented: *modules* and *configurations*. Modules are implemented by application code (written in a C-like syntax). Configurations are implemented by connecting interfaces of existing components.

The implementation part of a module is written in C-like code. A command or an event bar in an interface `foo` is referred as `foo.bar`. A keyword `call` indicates the invocation of a command. A keyword `signal` indicates the triggering by an event. For example, Figure 7.8 shows part of the implementation of the `Timer` component, whose interface is defined in Figure 7.7. In a sense, this implementation is very much like an object in object-oriented programming without any constructors.

Configuration is another kind of implementation of components, obtained by connecting existing components. Suppose we want to connect the `Timer` component and a hardware clock wrapper, called `HWClock`, to provide a timer service, called `TimerC`. Figure 7.9 shows a conceptual diagram of how the components are connected, and Figure 7.10 shows the corresponding `nesC` code.

```

    module Timer {
        provides {
interface StdControl;
interface Timer01;
        }
    uses interface Clock as Clk;
    }
    implementation {
        bool evenFlag;

command result_t StdControl.init      ()  {
    evenFlag = 0;
    return call Clk.setRate(128, 4); //4 ticks per second
    }
event result_t Clk.fire() {
    evenFlag = !evenFlag;
    if (evenFlag) {
    signal Timer01.timer0Fire();
    }
    }
    }

```

```
} else {  
signal Timer01.timer1Fire();  
}  
return SUCCESS;  
}  
...  
}
```

Figure 7.8 The implementation definition of the Timer component in nesC.

First of all, notice that the keyword configuration in the specification indicates that this component is not implemented directly as a module. In the implementation section of the configuration, the code first includes the two components, and then specifies that the interface StdControl of the TimerC component is the StdControl interface of the TimerModule; similarly for the Timer01 interface. The connection between the Clock interfaces is specified using the -> operator. Essentially, this interface is hidden from upper layers.

Figure 7.9 The TimerC configuration implemented by connecting Timer with HWClock.

Recall that TinyOS does not support dynamic memory allocation, so all components are statically constructed at compile time.

A complete application is always a configuration rather than a module. An application must contain the Main module, which links the code to the scheduler at run time. The Main has a single StdControl interface, which is the ultimate source of initialization of all components.

Concurrency and Atomicity

The language nesC directly reflects the TinyOS execution model through the notion of command and event contexts. Figure 7.11

```

configuration TimerC {
  provides {
    interface StdControl;
    interface Timer01;
  }
}
implementation {
  components TimerModule, Clock;

  StdControl = TimerModule.StdControl;
  Timer = TimerModule.Timer;

  TimerModule.Clk -> HWClock.Clock;
}

```

Figure 7.10 The implementation definition of the TimerC configuration in nesC.

shows a section of the component SenseAndSend to illustrate some language features to support concurrency in nesC and the effort to reduce race conditions. The SenseAndSend component is intended to be built on top of the Timer component (described in the previous section), an ADC component, which can provide sensor readings, and a communication component, which can send (or, more pre-cisely, broadcast) a packet. When responding to a timer0Fire event, the SenseAndSend component invokes the ADC to poll a sensor reading. Since polling a sensor reading can take a long time, a split-phase operation is implemented for getting sensor readings. The call to ADC.getData() returns immediately, and the completion of the operation is signaled by an ADC.dataReady() event. A busy flag is used to explicitly reject new requests while the ADC is fulfilling an exist-ing request. The ADC.getData() method sets the flag to true, while the ADC.dataReady() method sets it back to false. Sending the sensor reading to the next-hop neighbor via wireless communication is also a long operation. To make sure that it does not block the processing

of the ADC.dataReady() event, a separate task is posted to the scheduler. A task is a method defined using the task keyword. In order

ANNAL WOMEN'S COLLEGE


```

module SenseAndSend{
  provides interface StdControl;
  uses interface ADC;
  uses interface Timer;
  uses interface Send;
}

implementation {
  bool busy;

  norace uint16_t sensorReading;

  command result_t StdControl.init() {
    busy = FALSE;
  }

  event result_t Timer.timer0Fire() {
    bool localBusy; atomic
    {
      localBusy = busy; busy =
      TRUE;
    }
    if (!localBusy) {
      call ADC.getData(); //start getting sensor reading return SUCCESS;
    } else { return FAILED;
    }
  }

  task void sendData() { // send sensorReading
    adcPacket.data = sensorReading;
    call Send.send(&adcPacket, sizeof adcPacket.data); return SUCCESS;
  }
  event result_t ADC.dataReady(uint16_t data) { sensorReading = data;
    post sendData(); atomic { busy = FALSE;
  }
  return SUCCESS;
  }
  ...
}

```

Figure 7.11 A section of the implementation of SenseAndSend, illustrating the handling of concurrency in nesC.

Wireless Sensor Networks S.Leelavathi

to simplify the data structures inside the scheduler, a task cannot have arguments. Thus the sensor reading to be sent is put into a sensorReading variable.

There is one source of race condition in the SenseAndSend, which is the updating of the busy flag. To prevent some state from being updated by both scheduled tasks and event-triggered

interrupt handlers, nesC provides language facilities to limit the race conditions among these operations.

In nesC, code can be classified into two types:

- *Asynchronous code (AC)*: Code that is reachable from at least one interrupt handler.
- *Synchronous code (SC)*: Code that is only reachable from tasks.

Because the execution of TinyOS tasks are nonpreemptive and interrupt handlers preempts tasks, SC is always atomic with respect to other SCs. However, any update to shared state from AC, or from SC that is also updated from AC, is a potential race condition. To rein-state atomicity of updating shared state, nesC provides a keyword `atomic` to indicate that the execution of a block of statements should not be preempted. This construction can be efficiently implemented by turning off hardware interrupts. To prevent blocking the interrupts for too long and affecting the responsiveness of the node, nesC does not allow method calls in atomic blocks. In fact, nesC has a compiler rule to enforce the accessing of shared variables to maintain the race-free condition. If a variable x is accessed by AC, then any access of x outside of an atomic statement is a compile-time error. This rule may be too rigid in reality. When a programmer knows for sure that a data race is not going to occur, or does not care if it occurs, then a `norace` declaration of the variable can prevent the compiler from checking the race condition on that variable.

Thus, to correctly handle concurrency, nesC programmers need to have a clear idea of what is synchronous code and what is asynchronous code. However, since the semantics is hidden away in the layered structure of TinyOS, it is sometimes not obvious to the programmers where to add atomic blocks.

7.3.3 Dataflow-Style Language: TinyGALS

Dataflow languages [3] are intuitive for expressing computation on interrelated data units by specifying data dependencies among them. A dataflow program has a set of processing units called *actors*. Actors have ports to receive and produce data, and the directional connections among ports are FIFO queues that mediate the flow of data. Actors in dataflow languages intrinsically capture concurrency in a system, and the FIFO queues give a structured way of decoupling their executions. The execution of an actor is triggered when there are enough input data at the input ports.

Asynchronous event-driven execution can be viewed as a special case of dataflow models, where each actor is triggered by every incoming event. The *globally asynchronous and locally synchronous* (GALS) mechanism is a way of building event-triggered concurrent execution from thread-unsafe components. TinyGALS is such a language for TinyOS.

One of the key factors that affects component reusability in embedded software is the component composability, especially concurrent composability. In general, when developing a component, a programmer may not anticipate all possible scenarios in which the component may be used. Implementing all access to variables as atomic blocks incurs too much overhead. At the other extreme, making all variable access unprotected is easy for coding but certainly introduces bugs in concurrent composition. TinyGALS addresses concurrency concerns at the system level, rather than at the component level as in nesC. Reactions to concurrent events are managed by a dataflow-style FIFO queue communication.

TinyGALS Programming Model

TinyGALS supports all TinyOS components, including its interfaces and module implementations.¹ All method calls in a component

¹ Although posting tasks is not part of the TinyGALS semantics, the TinyGALS compiler and run time are compatible with it.

interface are synchronous method calls—that is, the thread of control enters immediately into the callee component from the caller component. An application in TinyGALS is built in two steps: (1) constructing asynchronous actors from synchronous components,⁵ and (2) constructing an application by connecting the asynchronous components through FIFO queues.

An actor in TinyGALS has a set of input ports, a set of output ports, and a set of connected TinyOS components. An actor is constructed by connecting synchronous method calls among TinyOS components. For example, Figure 7.12 shows a construction of TimerActor

Figure 7.12 Construction of a TimerActor from a Timer component and a Clock component.

In the implementation of TinyGALS as described in [39], which is based on TinyOS 0.6.1 and predates nesC, the asynchronous actors are called *modules*, and asynchronous connections are represented as “->”. To avoid the confusion with nesC, we have modified some of the TinyGALS syntax for inclusion in this section.

Chapter 7 Sensor Network Platforms and Tools

from two TinyOS components (i.e., nesC modules), Timer and Clock. Figure 7.13 is the corresponding TinyGALS code. An actor can expose one or more initialization methods. These methods are called by the TinyGALS run time before the start of an application. Initialization methods are called in a nondeterministic order, so their implementations should not have any cross-component dependencies.

At the application level, the asynchronous communication of actors is mediated using FIFO queues. Each connection can be parameterized by a queue size. In the current implementation of TinyGALS, events are discarded when the queue is full. However, other mechanisms such as discarding the oldest event can be used. Figure 7.14 shows a TinyGALS composition of timing, sensing, and sending part of the FieldMonitor application in Figure 7.5.

```
Actor TimerActor {
include components {
TimerModule;
HWClock;
}
init {
TimerModule.init;
}
port in { timerStart;
}
}
port out {
zeroFire; oneFire;
}
}
implementation {
timerStart -> TimerModule.Timer.start;
TimerModule.Clk -> HWClock.Clock;
```

```

TimerModule.Timer.timer0Fire    -> zeroFire;
TimerModule.Timer.timer1Fire    -> oneFire;
}
    
```

Figure 7.13 Implementation of the TimerActor in TinyGALS.

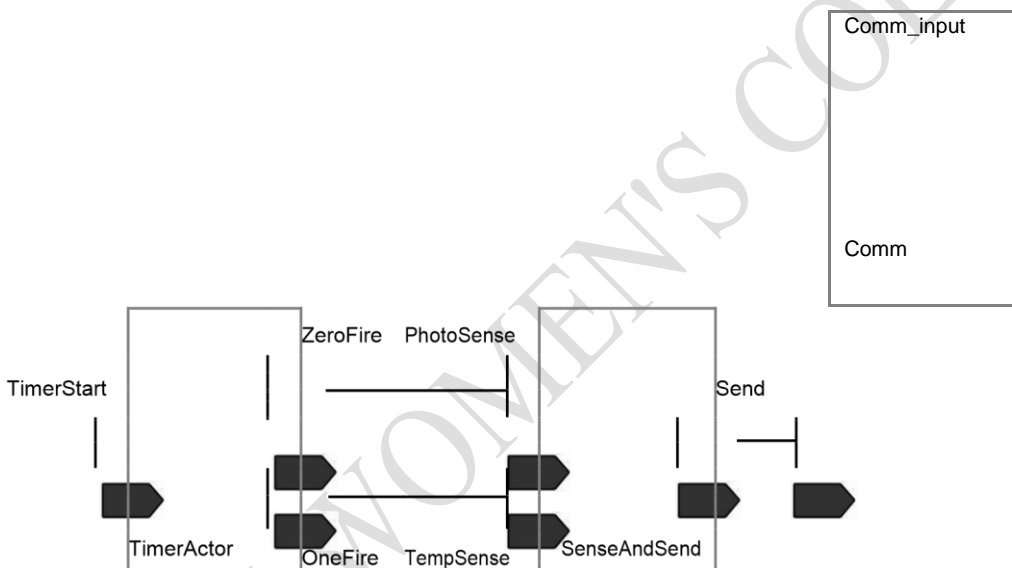


Figure 7.14 Triggering, sensing, and sending actors of the FieldMonitor in TinyGALS.

```

Application FieldMonitor {
include actors {    TimerActor;

SenseAndSend;
Comm;
}
implementation {
    zeroFire =>
photoSense 5;    oneFire =>
tempSense 5; send =>
comm_input 10;
}
    
```

```
}  
START@ timerStart;  
}
```

Figure 7.15 Implementation of the FieldMonitor in TinyGALS.

Figure 7.15 is the TinyGALS specification of the configuration in Figure 7.14. We omit the details of the SenseAndSend actor and the Comm actor, whose ports are shown in Figure 7.14. The symbol => represents a FIFO queue connecting input ports and output ports. The integer at the end of the line specifies the queue size. The command START@ indicates that the TinyGALS run time puts an initial event into the corresponding port after all initialization is finished. In our example, an event inserted into the timerStart port starts the HWClock, and the rest of the execution is driven by clock interrupt events.

The TinyGALS programming model has the advantage that actors become decoupled through message passing and are easy to develop independently. However, each message passed will trigger the scheduler and activate a receiving actor, which may quickly become

inefficient if there is a global state that must be shared among multiple actors. TinyGUYS (Guarded Yet Synchronous) variables are a mechanism for sharing global state, allowing quick access but with protected modification of the data.

In the TinyGUYS mechanism, global variables are guarded. Actors may read the global variables synchronously (without delay). However, writes to the variables are asynchronous in the sense that all writes are buffered. The buffer is of size one, so the last actor that writes to a variable wins. TinyGUYS variables are updated by the scheduler only when it is safe (e.g., after one module finishes and before the scheduler triggers the next module).

TinyGUYS have global names defined at the application level which are mapped to the parameters of each actor and are further mapped to the external variables of the components that use these variables. The external variables are accessed within a component by using special keywords: `PARAM_GET` and `PARAM_PUT`. The code generator produces thread-safe implementation of these methods using locking mechanisms, such as turning off interrupts.

TinyGALS Code Generation

TinyGALS takes a generative approach to mapping high-level constructs such as FIFO queues and actors into executables on Berkeley notes. Given the highly structured architecture of TinyGALS applications, efficient scheduling and event handling code can be automatically generated to free software developers from writing error-prone concurrency control code. The rest of this section discusses a code generation tool that is implemented based on TinyOS v0.6.1 for Berkeley notes.

Given the definitions for the components, actors, and application, the code generator automatically generates all of the necessary code for (1) component links and actor connections, (2) application initialization and start of execution, (3) communication among actors, and (4) global variable reads and writes.

Similar to how TinyOS deals with connected method calls among components, the TinyGALS code generator generates a set of aliases for each synchronous method call. The code generator also creates a system-level initialization function called `app_init()`, which contains calls to the `init()` method of each actor in the system. The `app_init()` function is one of the first functions called by the TinyGALS run-time scheduler before executing the application. An application start function `app_start()` is created based on the `@start` annotation. This function triggers the input port of the actor defined as the application starting point.

The code generator automatically generates a set of scheduler data structures and functions for each asynchronous connection between actors. For each input port of an actor, the code generator generates a queue of length n , where n is specified in the application definition. The width of the queue depends on the number of arguments of the method connected to the port. If

there are no arguments, then as an optimization, no queue is generated for the port (but space is still reserved for events in the scheduler event queue).

For each output port of an actor, the code generator generates a function that has the same name as the output port. This function is called whenever a method of a component wishes to write to an output port. The type signature of the output port function matches that of the method that connects to the port. For each input port connected to the output port, a `put()` function is generated which handles the actual copying of data to the input port queue. The output port function calls the input port's `put()` function for each connected input port. The `put()` function adds the port identifier to the scheduler event queue so that the scheduler will activate the actor at a later time.

For each connection between a component method and an actor input port, a function is generated with a name formed from the name of the input port and the name of the component method. When the scheduler activates an actor via an input port, it first calls this generated function to remove data from the input port queue and then passes it to the component method.

For each TinyGUYS variable declared in the application definition, a pair of data structures and a pair of access functions are generated. The pair of data structures consists of a data storage location of the type specified in the module definition that uses the global variable, along with a buffer for the storage location. The pair of access functions consists of a `PARAM_GET()` function that returns the value of the global variable, and a `PARAM_PUT()` function that stores a new value for the variable in the variable's buffer. A generated flag indicates whether the scheduler needs to update the variables by copying data from the buffer.

Since most of the data structures in the TinyGALS run-time scheduler are generated, the scheduler does not need to worry about handling different data types and the conversion among them. What is left in the run-time scheduler is merely event-queuing and function-triggering mechanisms. As a result, the TinyGALS run-time scheduler is very lightweight. The scheduler itself takes 112 bytes of memory, comparable with the original 86-byte TinyOS v0.6.1 scheduler.

7.4 Node-Level Simulators

Node-level design methodologies are usually associated with simulators that simulate the behavior of a sensor network on a per-node basis. Using simulation, designers can quickly study the performance (in terms of timing, power, bandwidth, and scalability) of potential algorithms without implementing them on actual hardware and dealing with the vagaries of actual physical phenomena. A node-level simulator typically has the following components:

Sensor node model: A node in a simulator acts as a software execution platform, a sensor host, as well as a communication terminal. In order for designers to focus on the application-level code, a node model typically provides or simulates a communication protocol stack, sensor behaviors (e.g., sensing noise), and operating system services. If the nodes are mobile,

then the positions and motion properties of the nodes need to be modeled. If energy characteristics are part of the design considerations, then the power consumption of the nodes needs to be modeled.

Communication model: Depending on the details of modeling, communication may be captured at different layers. The most elaborate simulators model the communication media at the physical layer, simulating the RF propagation delay and collision of simultaneous transmissions. Alternately, the communication may be simulated at the MAC layer or network layer, using, for example, stochastic processes to represent low-level behaviors.

Physical environment model: A key element of the environment within which a sensor network operates is the physical phenomenon of interest. The environment can also be simulated at various levels of detail. For example, a moving object in the physical world may be abstracted into a point signal source. The motion of the point signal source may be modeled by differential equations or interpolated from a trajectory profile. If the sensor network is passive—that is, it does not impact the behavior of the environment—then the environment can be simulated separately or can even be stored in data files for sensor nodes to read in. If, in addition to sensing, the network also performs actions that influence the behavior of the environment, then a more tightly integrated simulation mechanism is required.

Statistics and visualization: The simulation results need to be collected for analysis. Since the goal of a simulation is typically to derive global properties from the execution of individual nodes, visualizing global behaviors is extremely important. An ideal visualization tool should allow users to easily observe on demand the spatial distribution and mobility of the nodes, the connectivity among nodes, link qualities, end-to-end communication routes and delays, phenomena and their spatio-temporal dynamics, sensor readings on each node, sensor node states, and node lifetime parameters (e.g., battery power).

A sensor network simulator simulates the behavior of a subset of the sensor nodes with respect to time. Depending on how the time is advanced in the simulation, there are two types of execution models: *cycle-driven simulation* and *discrete-event simulation*. A cycle-driven (CD) simulation discretizes the continuous notion of real time into (typically regularly spaced) ticks and simulates the system behavior at these ticks. At each tick, the physical phenomena are first simulated, and then all nodes are checked to see if they have anything to sense, process, or communicate. Sensing and computation are assumed to be finished before the next tick. Sending a packet is also assumed to be completed by then. However, the packet will not be available for the destination node until the next tick. This split-phase communication is a key mechanism to reduce cyclic dependencies that may occur in cycle-driven simulations. That is, there should be no two components, such that one of them computes $yk = f(xk)$ and the other computes $xk = g(yk)$, for the same tick index k . In fact, one of the most subtle issues in designing a CD simulator

is how to detect and deal with cyclic dependencies among nodes or algorithm components. Most CD simulators do not allow interdependencies within a single tick. Synchronous languages [91], which are typically used in control system designs rather than sensor network designs, do allow cyclic dependencies. They use a fixed-point semantics to define the behavior of a system at each tick.

Unlike cycle-driven simulators, a discrete-event (DE) simulator assumes that the time is continuous and an event may occur at any time. An event is a 2-tuple with a value and a time stamp indicating when the event is supposed to be handled. Components in a DE simulation react to input events and produce output events. In node-level simulators, a component can be a sensor node and the events can be communication packets; or a component can be a software module within a node and the events can be message passings among these modules. Typically, components are *causal*, in the sense that if an output event is computed from an input event, then the time stamp of the output event should not be earlier than that of the input event. Noncausal components require the simulators to be able to roll back in time, and, worse, they may not define a deterministic behavior of a system [129]. A DE simulator typically requires a global event queue. All events passing between nodes or modules are put in the event queue and sorted according to their chronological order. At each iteration of the simulation, the simulator removes the first event (the one with the earliest time stamp) from the queue and triggers the component that reacts to that event.

In terms of timing behavior, a DE simulator is more accurate than a CD simulator, and, as a consequence, DE simulators run slower. The overhead of ordering all events and computation, in addition to the values and time stamps of events, usually dominates the computation time. At an early stage of a design when only the asymptotic behaviors rather than timing properties are of concern, CD simulations usually require less complex components and give faster simulations. Partly because of the approximate timing behaviors, which make simulation results less comparable from application to application, there is no general CD simulator that fits all sensor network simulation tasks. We have come across a number of home-grown simulators written in Matlab, Java, and C++. Many of them are developed for particular applications and exploit application-specific assumptions to gain efficiency.

DE simulations are sometimes considered as good as actual implementations, because of their continuous notion of time and discrete notion of events. There are several open-source or commercial simulators available. One class of these simulators comprises extensions of classical network simulators, such as ns-2,⁶ J-Sim (previously known as JavaSim),⁷ and GloMoSim/QualNet.⁸ The focus of these simulators is on network modeling, protocols stacks, and simulation performance. Another class of simulators, sometimes called *software-in-the-loop simulators*, incorporate the actual node software into the simulation. For this reason, they are typically attached to particular hardware platforms and are less portable. Examples include TOSSIM [131] for Berkeley motes and Em* (pronounced *em star*) [62] for Linux-based nodes such as Sensoria WINS NG platforms.

7.4.1 The ns-2 Simulator and its Sensor Network Extensions

The simulator ns-2 is an open-source network simulator that was originally designed for wired, IP networks. Extensions have been made

Available at <http://www.isi.edu/nsnam/ns>.

⁷ Available at <http://www.j-sim.org>.

⁸ Available at <http://pcl.cs.ucla.edu/projects/glomosim>.

to simulate wireless/mobile networks (e.g., 802.11 MAC and TDMA MAC) and more recently sensor networks. While the original ns-2 only supports logical addresses for each node, the wireless/mobile extension of it (e.g., [25]) introduces the notion of node locations and a simple wireless channel model. This is not a trivial extension, since once the nodes move, the simulator needs to check for each physical layer event whether the destination node is within the communication range. For a large network, this significantly slows down the simulation speed.

There are at least two efforts to extend ns-2 to simulate sensor networks: SensorSim from UCLA⁹ and the NRL sensor network extension from the Navy Research

Laboratory.¹⁰ SensorSim aims at providing an energy model for sensor nodes and communication, so that power properties can be simulated [175]. SensorSim also supports hybrid simulation, where some real sensor nodes, running real applications, can be executed together with a simulation. The NRL sensor network extension provides a flexible way of modeling physical phenomena in a discrete event simulator. Physical phenomena are modeled as network nodes which communicate with real nodes through physical layers. Any interesting events are sent to the nodes that can sense them as a form of communication. The receiving nodes simply have a sensor stack parallel to the network stack that processes these events.

The main functionality of ns-2 is implemented in C++, while the dynamics of the simulation (e.g., time-dependent application characteristics) is controlled by Tcl scripts. Basic components in ns-2 are the layers in the protocol stack. They implement the *handlers* interface, indicating that they handle events. Events are communication packets that are passed between consecutive layers within one node, or between the same layers across nodes.

The key advantage of ns-2 is its rich libraries of protocols for nearly all network layers and for many routing mechanisms. These protocols

Available at <http://nesl.ee.ucla.edu/projects/sensorsim/>.

Available at <http://pf.itd.nrl.navy.mil/projects/nrlsensorsim/>.

are modeled in fair detail, so that they closely resemble the actual protocol implementations. Examples include the following:

TCP: reno, tahoe, vegas, and SACK implementations

MAC: 802.3, 802.11, and TDMA

Ad hoc routing: Destination sequenced distance vector (DSDV) routing, dynamic source routing (DSR), ad hoc on-demand distance vector (AODV) routing, and temporally ordered routing algorithm (TORA)

Sensor network routing: Directed diffusion, geographical routing (GEAR) and geographical adaptive fidelity (GAF) routing.

7.4.2 The Simulator TOSSIM

TOSSIM is a dedicated simulator for TinyOS applications running on one or more Berkeley notes. The key design decisions on building TOSSIM were to make it scalable to a network of potentially thousands of nodes, and to be able to use the actual software code in the simulation. To achieve these goals, TOSSIM takes a cross-compilation approach that compiles the nesC source code into components in the simulation. The event-driven execution model of TinyOS greatly simplifies the design of TOSSIM. By replacing a few low-level components, such as the A/D conversion (ADC), the system clock, and the radio front end, TOSSIM translates hardware interrupts into discrete event simulator events. The simulator event queue delivers the interrupts that drive the execution of a node. The upper-layer TinyOS code runs unchanged.

TOSSIM uses a simple but powerful abstraction to model a wireless network. A network is a *directed* graph, where each vertex is a sensor node and each directed edge has a bit-error rate. Each node has a private piece of state representing what it hears on the radio channel. By setting connections among the vertices in the graph and a bit-error rate on each connection, wireless channel characteristics, such as imperfect channels, hidden terminal problems, and asymmetric links, can be easily modeled. Wireless transmissions are simulated at the bit level. If a bit error occurs, the simulator flips the bit.

TOSSIM has a visualization package called TinyViz, which is a Java application that can connect to TOSSIM simulations. TinyViz also provides mechanisms to control a running

simulation by, for example, modifying ADC readings, changing channel properties, and injecting packets. TinyViz is designed as a communication service that interacts with the TOSSIM event queue. The exact visual interface takes the form of plug-ins that can interpret TOSSIM events. Beside the default visual interfaces, users can add application-specific ones easily.

7.5 Programming Beyond Individual Nodes: State-Centric Programming

Many sensor network applications, such as target tracking, are not simply generic distributed programs over an ad hoc network of energy-constrained nodes. Deeply rooted in these applications is the notion of states of physical phenomena and models of their evolution over space and time. Some of these states may be represented on a small number of nodes and evolve over time, as in the target tracking problem in Chapter 2, while others may be represented over a large and spatially distributed number of nodes, as in tracking a temperature contour.

A distinctive property of physical states, such as location, shape, and motion of objects, is their continuity in space and time. Their sensing and control is typically done through sequential state updates. System theories, the basis for most signal and information processing algorithms, provide abstractions for state update, such as:

$$\mathbf{x}_{k+1} = f(\mathbf{x}_k, u_k) \quad (7.1) \quad \mathbf{y}$$

$$y_k = g(\mathbf{x}_k, u_k) \quad (7.2)$$

where \mathbf{x} is the state of a system, u are the inputs, \mathbf{y} are the outputs, k is an integer update index over space and/or time, f is the state update function, and g is the output or observation function. This formulation is broad enough to capture a wide variety of algorithms in sensor fusion, signal processing, and control (e.g., Kalman filtering, Bayesian estimation, system identification, feedback control laws, and finite-state automata).

However, in distributed real-time embedded systems such as sensor networks, the formulation is not so clean as represented in those equations. The relationships among subsystems can be highly complex and dynamic over space and time. The following concerns, not explicitly raised (7.1) and (7.2), must be properly addressed during the design to ensure the correctness and efficiency of the resulting systems.

Where are the state variables stored?

Where do the inputs come from?

Where do the outputs go?

Where are the functions f and g evaluated?

How long does the acquisition of inputs take?

Are the inputs in uk collected synchronously?

Do the inputs arrive in the correct order through communication?

What is the time duration between indices k and $k + 1$? Is it a constant?

These issues, addressing *where* and *when*, rather than *how*, to perform sensing, computation, and communication, play a central role in the overall system performance. However, these “nonfunctional” aspects of computation, related to concurrency, responsiveness, networking, and resource management, are not well supported by traditional programming models and languages. State-centric programming aims at providing design methodologies and frameworks that give meaningful abstractions for these issues, so that system designers can continue to write algorithms like (7.1) and (7.2) on

top of an intuitive understanding of where and when the operations are performed. This section introduces one such abstraction, namely, collaboration groups.

7.5.1 Collaboration Groups

A collaboration group is a set of entities that contribute to a state update. These entities can be physical sensor nodes, or they can be more abstract system components such as virtual sensors or mobile agents hopping among sensors. In this context, they are all referred to as *agents*.

Intuitively, a collaboration group provides two abstractions: its *scope* to encapsulate network topologies and its *structure* to encapsulate communication protocols. The scope of a group defines the membership of the nodes with respect to the group. For the discussion of collaboration groups in this chapter, we broaden the notion of nodes to include both physical sensor nodes and virtual sensor nodes that may not be attached to any physical sensor. In this broader sense of node, a software agent that hops among the sensor nodes to track a target is a virtual node. Limiting the scope of a group to a sub-set of the entire space of all agents improves scalability. The scope of a group can be specified existentially or by a membership function (e.g., all nodes in a geometric extent, all nodes within a certain number of hops from an anchor node, or all nodes that are “close enough” to a temperature contour). Grouping nodes according to some physical attributes rather than node addresses is an important and distinguishing characteristic of sensor networks.

The *structure* of a group defines the “roles” each member plays in the group, and thus the flow of data. Are all members in the group equal peers? Is there a “leader” member in the group that consumes data? Do members in the group form a tree with parent and children relations? For example, a group may have a leader node that collects certain sensor readings from all followers. By mapping the leader and the followers onto concrete sensor nodes, we effectively define the flow of data from the hosts of followers to the host of the leader. The notion of roles also shields programmers from addressing individual

nodes either by name or address. Furthermore, having multiple members with the same role provides some degree of redundancy and improves robustness of the application in the presence of node and link failures.

Formally, a group is a 4-tuple:

$$= (A, L, p, R)$$

where

A is a set of agents;

L is a set of labels, called *roles*;

$p : A \rightarrow L$ is a function that assigns each agent a role;

$R \subseteq L \times L$ are the connectivity relations among roles.

Given the relations among roles, a group can induce a lower-level connectivity relation E among the agents, so that $\forall a, b \in A$, if

$p(a), p(b) \in R$, then $(a, b) \in E$. For example, under this formulation, the leader-follower structure defines two roles, $L = \{leader, follower\}$, and a connectivity relation, $R = \{(follower, leader)\}$, meaning that the follower sends data to the leader. Then, by specifying one leader agent and multiple follower agents within a geographical region (i.e., specifying a map p from a set of agents in A to labels in L), we have effectively specified that all followers send data to the leader without addressing the followers individually.

At run time, the scope and structural dynamics of groups are managed by group management protocols, which are highly dependent on the types of groups. A detailed specification of group management protocols is beyond the scope of this section. Some examples of these protocols are discussed here at a high level. Interested readers can refer to Chapter 3 for more detail.

Examples of Groups

Combinations of scopes and structures create patterns of groups that may be highly reusable from application to application. Here, we give several examples of groups, though by no means is it a complete list. The goal is to illustrate the wide variety of the kinds of groups, and the importance of mixing and matching them in applications.

Geographically Constrained Group. A geographically constrained group (GCG) consists of members within a prespecified geographical extent. Since physical signals, especially the ones from point targets, may propagate only to a limited extent in an environment, this kind of group naturally represents all the sensor nodes that can possibly “sense” a phenomenon. There are many ways to specify the geographic shape, such as circles, polygons, and their unions and intersections. A GCG can be easily established by geographically constrained flooding. Protocols such as Geocasting [117], GEAR [229], and Mobicast [102] may be used to support the communication among members even in the

presence of communication “holes” in the region. A GCG may have a leader, which fuses information from all other members in the group.

***n*-hop Neighborhood Group.** When the communication topology is more important than the geographical extent, hop counts are useful to constrain group membership. An *n*-hop neighborhood group (*n*-HNG) has an anchor node and defines that all nodes within *n* communication hops are members of the group. Since it uses hop counts rather than Euclidean distances, local broadcasting can be used to determine the scope. Usually, the anchor node is the leader of the group, and the group may have a tree structure with the leader as the root to optimize for communication. If the leader’s behavior can be decomposed into suboperations running on each node, then the tree structure also provides a platform for distributing the computation.

There are several useful special cases for *n*-HNG. For example, 0-HNG contains only the anchor node itself, 1-HNG comprises the one-hop neighbors of the anchor node, and ∞ -HNG contains all the nodes reachable from the root. From this point of view, TinyDB [149] (as discussed in Chapter 6) is built on a ∞ -HNG group.

Publish/Subscribe Group. A group may also be defined more dynamically, by all entities that can provide certain data or services, or that can satisfy certain predicates over their observations or internal states. A publish/subscribe group (PSG) comprises consumers expressing interest in specific types of data or services and producers that provide those data or services. Communication among members of a PSG may be established via rendezvous points, directory servers, or network protocols such as directed diffusion.

Acquaintance Group. An even more dynamic kind of group is the acquaintance group (AG), where a member belongs to the group because it was “invited” by another member in the group. The relationships among the members may not depend on any physical properties at the current time but may be purely logical and historical. A member may also quit the group without requiring permission from any other member. An AG may have a leader, serving as the rendezvous point. When the leader is also fixed on a node or in a region, GPSR [112], ad hoc routing trees, or directed diffusion types of protocols may facilitate the communication between the leader and the other members. An obvious use of this group is to monitor and control mobile agents from a base station. When all members in the group are mobile, there is no leader member, and any member may wish to communicate to one or more other members, the maintenance of connectivity among the group members can be nontrivial. The roaming hub (RoamHBA) protocol is an example of maintaining connectivity among mobile agents [67].

Using Multiple Types of Groups

Mixing and matching groups is a powerful technique for tackling system complexity by making algorithms much more scalable and resource efficient without sacrificing conceptual clarity. One may use highly tuned communication protocols for specific groups to reduce latency and energy costs.

There are various ways to compose groups. They can be composed in parallel to provide different types of input for a single computational entity. For example, in the target tracking problem in

7.5.2 PIECES: A State-Centric Design Framework

PIECES (Programming and Interaction Environment for Collaborative Embedded Systems) [141] is a software framework that implements the methodology of state-centric programming over collaboration groups to support the modeling, simulation, and design of sensor network applications. It is implemented in a mixed Java-Matlab environment.

Principals and Port Agents

PIECES comprises *principals* and *port agents*. Figure 7.16 shows the basic relations among principals and port agents.

A principal is the key component for maintaining a piece of *state*. Typically, a principal maintains state corresponding to certain aspects of the physical phenomenon of interest.¹¹ The role of a principal is to update its state from time to time, a computation corresponding to evaluating function f in (7.1). A principal also accepts other principals' queries of certain views on its own state, a computation corresponding to evaluating function g in (7.2).

To update its portion of the state, a principal may gather information from other principals. To achieve this, a principal creates port agents and attaches them onto itself and onto the other principals. A port agent may be an input, an output, or both. An output port

From a computational perspective, a port agent as an object certainly has its own state. But the distinction here is that the states of port agents are *not* about physical phenomena.

Figure 7.16 Principal and port agents (adapted from [141]).

agent is also called an *observer*, since it computes outputs based on the host principal's state and sends them to other agents. Observers may be active or passive. An active observer pushes data autonomously to its destination(s), while a passive observer sends data only when a consumer requests it. A principal typically attaches a set of observers to other principals and creates a local input port agent to receive the information collected by the remote agents. Thus port agents capture communication patterns among principals.

The execution of principals and port agents can be either time-driven or event-driven, where events may include physical events that are pushed to them (i.e., data-driven) or query events from other principals or agents (i.e., demand-driven). Principals maintain state, reflecting the physical phenomena. These states can be updated, rather than rediscovered, because the underlying physical states are typically continuous in time. How often the principal states need to be updated depends on the dynamics of the phenomena or physical events. The executions of observers, however, reflect the demands of the outputs. If an output is not currently needed, there is no need to compute it. The notion of "state" effectively separates these two execution flows.

To ensure consistency of state update over a distributed computational platform, PIECES requires that a piece of state, say $\mathbf{x}|s$, can only be maintained by exactly one principal. Note that this does not prevent other principals from having local caches of $\mathbf{x}|s$ for efficiency and performance reasons; nor does it prevent the other principals from locally updating the values of cached $\mathbf{x}|s$. However, there is only one "master copy" for $\mathbf{x}|s$; all local updates should be treated as "suggestions" to the master copy, and only the

principal that owns $x|s$ has the final word on its values. This asymmetric access of variables simplifies the way shared variables are managed.

Principal Groups

Principals can form groups. A principal group gives its members a means to find other relevant principals and attaches port agents to them. A principal may belong to multiple groups. A port agent, however, serving as a proxy for a principal in the group, can only be associated with one group.

The creation of groups can be delegated to port agents, especially for leader-based groups. The leader port agent, typically of type input, can be created on a principal, and the port agent can take group scope and structure parameters to find the other principals and create follower port agents on them. Groups can be created dynamically, based on the collaboration needs of principals. For example, when a tracking principal finds that there is more than one target in its sensing region, it may create a classification group to fulfill the need of classifying the targets. A group may have a limited time span. When certain collaborations are no longer needed, their corresponding groups can be deleted.

The structure of a group allows its members to address other principals through their role, rather than their name or logical address. For example, the only interface that a follower port agent in a leader-follower structured group needs is to send data to the leader. If the leader moves to another node while a data packet is moving from a follower agent to the leader, the group management protocol should take care of the dangling packet, either delivering it to the leader at the new location or simply discarding it. The group management protocol may be built on top of data-centric routing and storage services such as diffusion routing and GHT (discussed in earlier chapters).

Mobility

A principal is hosted by a specific network node at any given time. The most primitive type of principal is a *sensing principal*, which is fixed to a sensor node. A sensing principal maintains a piece of (local) state related to the physical phenomenon, based solely on its own local measurement history. Although a sensing principal is constrained to a physical node, other principals may be implemented as software agents that move from host to host, depending on information utility, performance requirements, time constraints, and resource availability. A principal P may also be *attached* to another principal Q in the sense that P moves with Q . When a principal moves, it carries its state to the new location and the scope of the group it belongs to may be updated if necessary.

Mobile principals bring additional challenges to maintaining the state. For example, a principal should not move while it is in the middle of updating the state. To ensure this, PIECES imposes the restriction that whenever an agent is triggered, its execution must have reached a quiescent state. Such a trigger is called a *responsible trigger* [147]. Only at these quiescent states can principals move to other nodes in a well-defined way, carrying a minimum amount of information representing the phenomena.

PIECES Simulator

PIECES provides a mixed-signal simulator that simulates sensor network applications at a high level. The simulator is implemented using a combination of Java and Matlab. An event-driven engine is built in Java to simulate network message passing and agent execution at the collaboration-group level. A continuous-time engine is built in Matlab to simulate target trajectories, signals and noise, and sensor front ends. The main control flow is in Java, which maintains the global notion of time. The interface between Java and Matlab also makes it possible to implement functional algorithms such as signal processing and sensor fusion in Matlab, while leaving their execution control in Java. A three-tier distributed architecture is designed through Java registrar and RMI interfaces, so that the execution in Java and Matlab can be separately interrupted and debugged.

Like most network simulators such as ns-2, the PIECES simulator maintains a global event queue and triggers computational entities—principals, port agents, and groups—via timed events. However, unlike network simulators that aim to accurately simulate network behavior at the packet level, the PIECES simulator verifies CSIP algorithms in a networked execution environment at the collaboration-group level. Although groups must have distributed implementations in real deployments, they are centralized objects in the simulator. They can internally make use of instant access to any member of any role, although these services are not available to either principals or port agents. This relieves the burden of having to develop, optimize, and test the communication protocols concurrently with the CSIP algorithms. The communication delay is estimated based on the locations of sender and receiver and the group management protocol being used. For example, if an output port of a sensing principal calls `sendToLeader(message)` on its container group, then the group determines the sensor nodes that host the sensing principal and the destination principal, computes the number of hops between the two nodes specified by the group management protocol, and generates a corresponding delay and a bit error based on the number of hops. A detailed example of using this simulator is given in the next section.

7.5.3 Multitarget Tracking Problem Revisited

Using the state-centric model, programmers decouple a global state into a set of independently maintained pieces, each of which is assigned a principal. To update the state, principals may look for inputs from other principals, with sensing principals supporting the lowest-level sensing and estimation tasks. Communication patterns are specified by defining collaboration groups over principals and assigning corresponding roles for each principal through port agents. A mobile principal may define a utility function, to be evaluated at candidate sensor nodes, and then move to the best next location, all in a way transparent to the application developer. Developers can focus on implementing the state update functions as if they are writing centralized programs.

To make these concepts concrete, let us revisit the multitarget tracking system introduced in Chapter 2. Recall that in Figure 2.5, the tracking of two crossing targets can be decomposed into three phases:

When the targets are far apart, the tracking problem can be treated as a set of singletarget tracking subproblems.

When the targets are in proximity of each other, they are tracked jointly due to signal mixing.

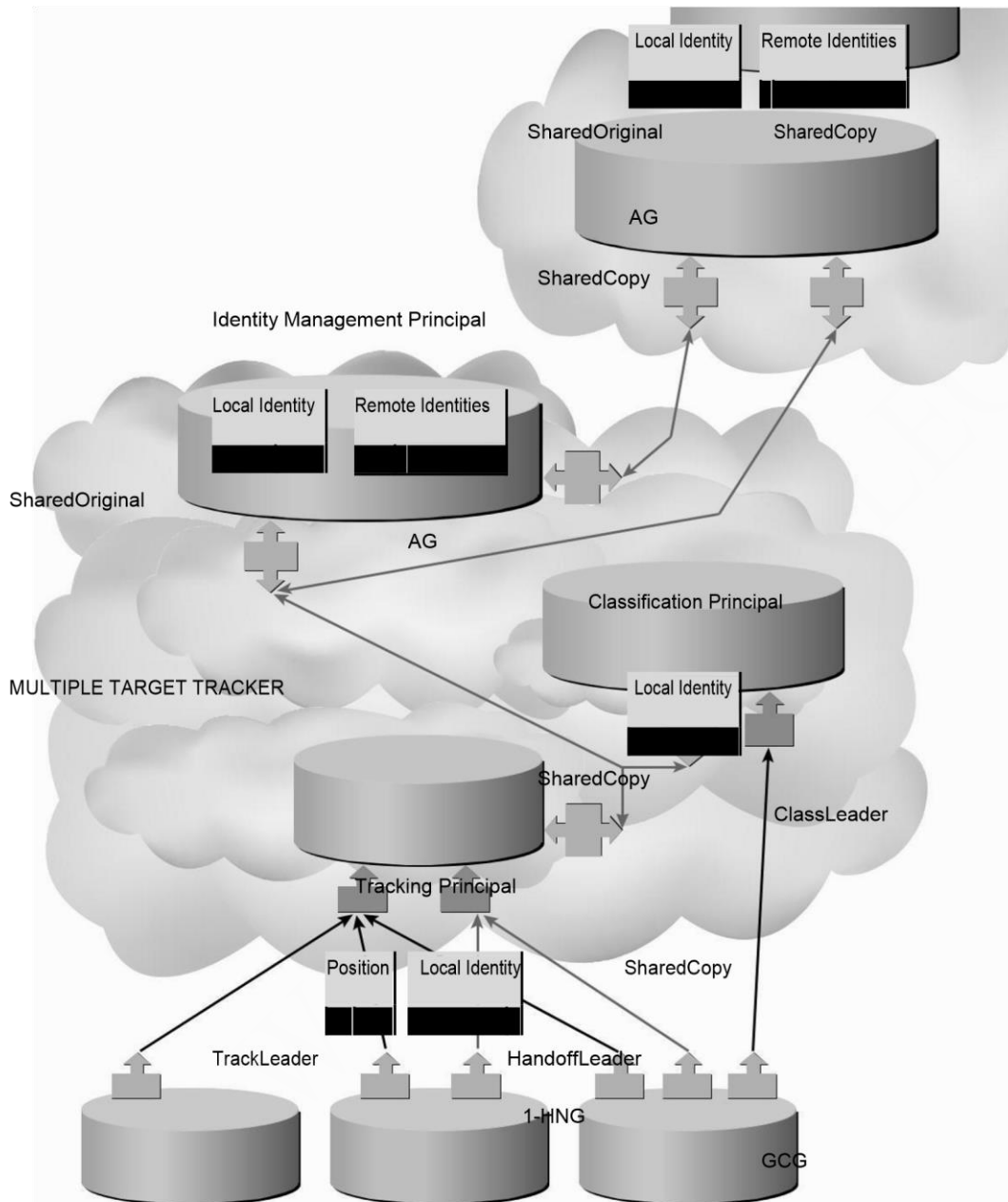
After the targets move apart, the tracking problem becomes two single-target tracking subproblems again.

To summarize, there are two kinds of target information that the user cares about in this context: target positions and target identities. In the third phase above, in addition to the problem of updating track locations, there is a need to sort out ambiguity regarding which track corresponds to which target. We refer to this problem as the *identity management* problem. Specifically, one must keep track of how the identities mix when targets cross over, and update identity information at the other node when credible target identity evidence is available to one node. The identity information may be obtained by a local classifier or by an identity management protocol across tracks. In PIECES, the system is designed as a set of communicating target trackers

(MTTrackers), where each tracker maintains the trajectory and identity information about a target or a set of spatially adjacent targets. An MTTracker is implemented by three principals: a *tracking principal*, a *classification principal*, and an *identity management principal*, as shown in Figure 7.17. In the first phase, the identity state of the track is trivial; thus no classification and identity management principals are needed.

ANNAI WOMEN'S COLLEGE

Identity Management Principal



TrackFollower TrackFollower TrackFollower HandoffFollower HandoffFollower
 Sensing Principal Sensing Principal Sensing Principal ClassFollower

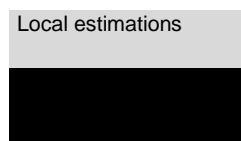
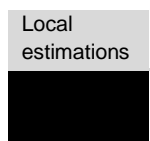
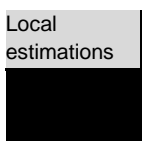


Figure 7.17 The distributed multi-object tracking algorithm as implemented in the state-centric programming model, using distributed principals and agents as discussed in the text. Notice that the state-centric model allows an application developer to focus on key pieces of state information the sensor network creates and maintains, thus raising the abstraction level of programming (adapted from [141]).

A tracking principal updates the track position state periodically. It collects local individual position estimates from sensors close to the target by a GCG with a leader-follower relation. The tracking principal is the leader, and all sensing principals within a certain geographical extent centered about the current target position estimate are the followers. The tracking principal also makes hopping decisions based on its current position estimate and the node characteristic information collected from its one-hop neighbors via a 1-HNG. When the principal is initialized, it creates the agents and corresponding groups. Behind the scene, the groups create follower agents with specific types of output, indicated by the sensor modalities. Without further instructions from the programmer, the followers periodically report their outputs to the input port agents. Whenever the leader principal is activated by a time trigger, it updates the target position using the newly received data from the followers and selects the next hosting node based on neighbor node characteristics

Both the classification principal and the identity management principal operate on the identity state, with the identity management principal maintaining the “master copy” of the state. In fact, the classification principal is created only when there is a need for classifying targets. The classification principal uses a GCG to collect class feature information from nearby sensing principals in the same way that tracking principals collect location estimates. The identity management principal forms an AG with all other identity management principals that may have relevant identity information. They become members of a particular identity group only when targets intersect and their identities mix. Both classification principals and identity management principals are *attached* to the tracking principal for their mobility decisions. However, the formation of an AG among these three principals also provides the flexibility that they can make their own hopping decisions without changing their interaction interface.

Simulation Results

Figure 7.18 shows the progression of tracking two crossing targets. Initially, when the targets are well separated, as in Figure 7.18(a)

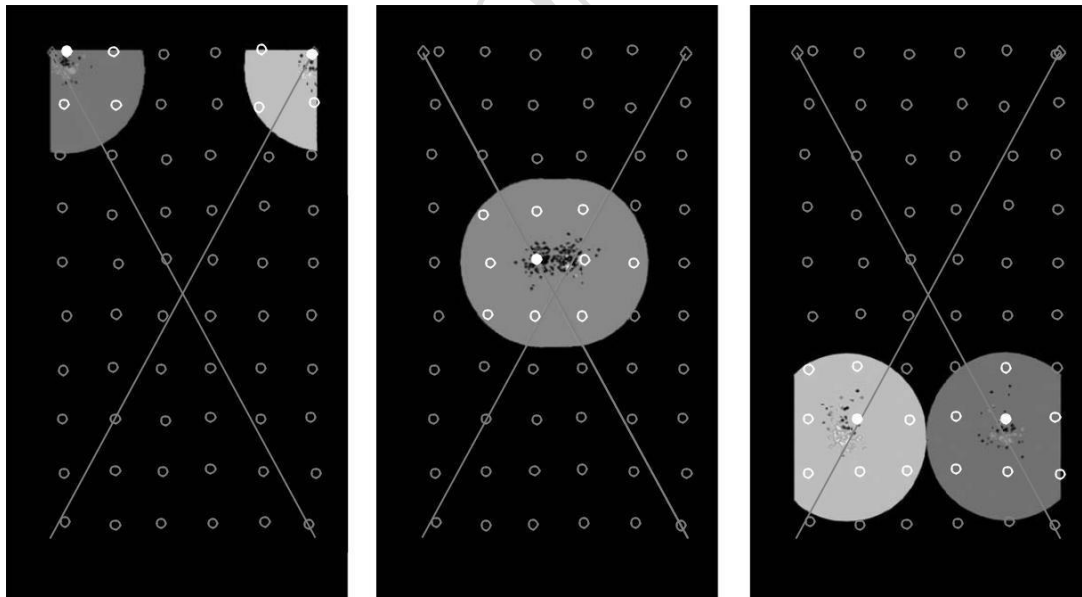


Figure 7.18 Simulation snapshots: Sensor nodes are indicated by small circles, and the crossing lines indicate the true trajectories of the two targets. One geographically constrained group is created for each target. When the two targets cross over, their groups merge into one.

each target is tracked by a tracker whose sensing group is pictured as a shaded disk. The hosting node of the tracking principal is plotted in solid white dots, and the hosts for corresponding sensing principals are plotted in small, empty white circles inside the shaded disks. Since the targets are well separated, each identity group contains only one member—the identity management principal of a tracker. As the targets move toward the center of the sensor field, the sensing groups move with their respective track positions. In Figure 7.18(b), the two separate tracking groups have merged. A joint tracking principal updates tracks for both targets. The reason for the merge is that when the two targets approach each other, it is more accurate to track the targets jointly, rather than independently, due to the effect of signal mixing. Finally, as the targets move away from each other, the merged tracking group splits into two separate single-target