

WIRELESS SENSOR NETWORKS

(18MCA55E)

UNIT - V

“SENSOR NETWORK PLATFORMS AND TOOLS”

FACULTY:

Dr. R. A. Roseline, M.Sc., M.Phil., Ph.D.,

Associate Professor and Head,

Post Graduate and Research Department of Computer Applications,



Government Arts College (Autonomous), Coimbatore – 641 018.

CONTENT

- Sensor Network Platforms and Tools
 - Sensor Node Hardware
 - Berkeley Motes
 - Sensor Network Programming Challenges
 - Node-Level Software Platforms
 - Operating System: TinyOS
 - Imperative Language: nesC
 - Dataflow-Style Language: TinyGALS
 - Node-Level Simulators
 - The ns-2 Simulator and its Sensor Network Extensions



SENSOR NETWORK PLATFORMS AND TOOLS



- Sensor networks, including sensing and estimation, networking, infrastructure services, sensor tasking, and data storage and query.
- 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.



- 
- 
- 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)
 - software comprises reactive, concurrent, distributed programs running on ad hoc, resource-constrained, unreliable computation and communication platforms. Developers at this level have to deal with all kinds of uncertainty in the real world.

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.
 1. **Augmented general-purpose computers:** Examples include low power PCs, embedded PCs (e.g., PC104), custom-designed PCs (e.g., Sensoria WINS NG nodes), 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.







- 
- 
- 2. Dedicated embedded sensor nodes:** 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 Tiny OS platform and its companion programming language, nesC.



- 
- 
- 3. 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 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.

BERKELEY MOTES

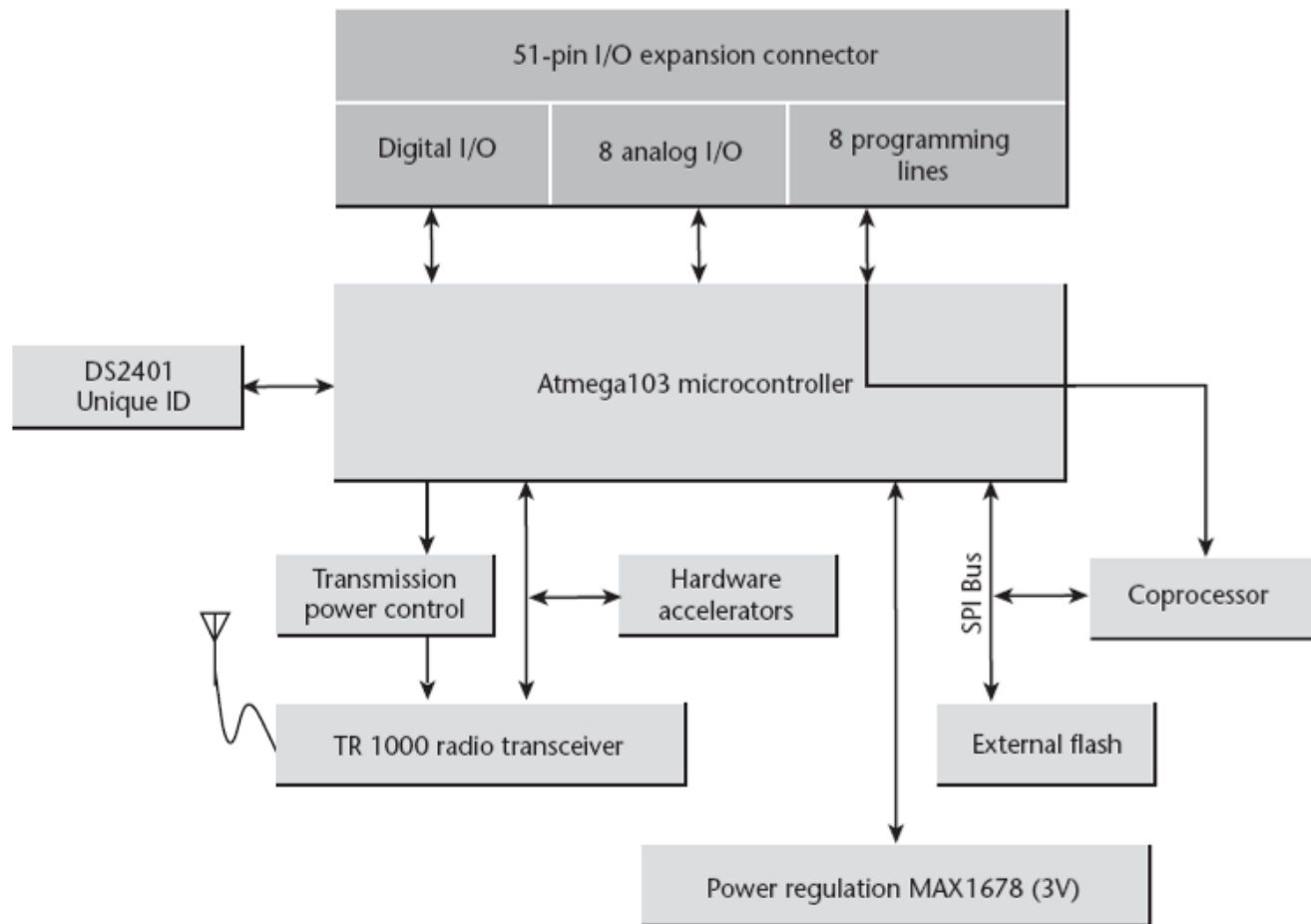
- The Berkeley motes are a family of embedded sensor nodes sharing roughly the same architecture.



A COMPARISON OF BERKELEY MOTES



Mote type		WeC	Rene	Rene2	Mica	Mica2	Mica2Dot
Example picture							
MCU	Chip	AT90LS8535	ATmega163L	ATmega103L	ATmega128L		
	Type	4 MHz, 8 bit	4 MHz, 8 bit	4 MHz, 8 bit	8 MHz, 8 bit		
	Program memory (KB)	8	16	128	128		
	RAM (KB)	0.5	1	4	4		
External nonvolatile storage	Chip	24LC256			AT45DB014B		
	Connection type	I2C			SPI		
	Size (KB)	32			512		
Default power source	Type	Coin cell	2xAA				Coin cell
	Typical capacity (mAh)	575	2850				1000
RF	Chip	TR1000				CC1000	
	Radio frequency	868/916MHz				868/916MHz, 433, or 315 MHz	
	Raw speed (kbps)	10			40	38.4	
	Modulation type	On/Off key			Amplitude Shift key	Frequency Shift key	

- 
- 
1. 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.
 2. Given these small memory sizes, writing software for motes is challenging. Ideally, programmers should be relieved from optimizing code at assembly level to keep code footprint small.
 3. However, high-level support and software 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.

MICA MOTE ARCHITECTURE





- 
- 
- 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

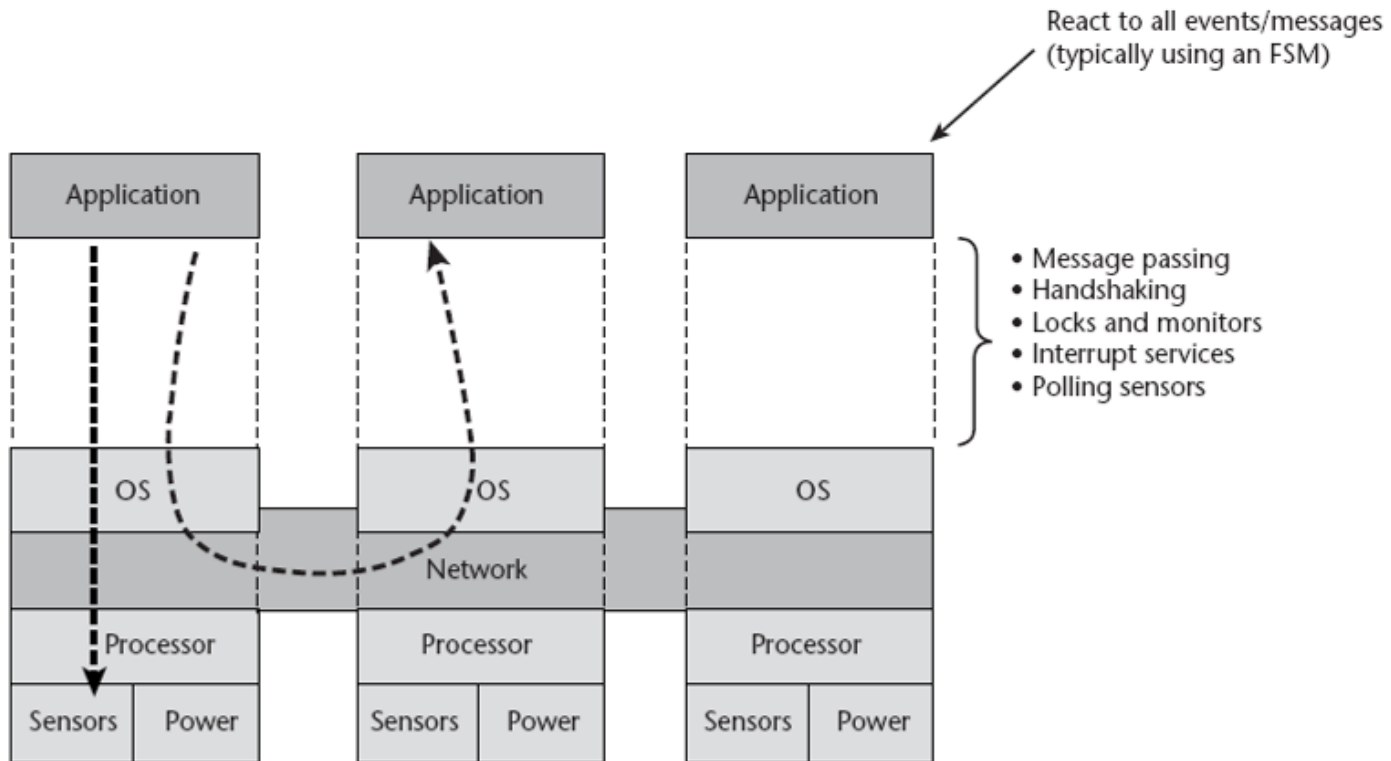
- 
- 
- MICA motes support a 51 pin I/O extension connector. Sensors, actuators, serial I/O boards, or parallel I/O boards can be connected via the connector.
 - A sensor/ actuator board can host a temperature sensor, a light sensor, an accelerometer, a magnetometer, a microphone, and a beeper.
 - The 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.

<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
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
Photoresister	2000 Hz	10 ms	1.235 mA
Accelerometer	100 Hz	10 ms	5 mA/axis
Temperature	2 Hz	500 ms	0.150 mA

POWER CONSUMPTION OF MICA MOTES



It is interesting to look at the energy consumption of various components on a MICA mote.



- 
- 
- A radio 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.







SENSOR NETWORK PROGRAMMING CHALLENGES

Traditional embedded system programming interface

- 
- 
- 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

- 
- 
- 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 modularize the operating system so that only the necessary parts are deployed with the application.
 - Real-time scheduling 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.



- 
- 
- 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.
 - 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.

NODE-LEVEL SOFTWARE PLATFORMS

- 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.
- Tiny OS and Tiny GALS are two representative examples of node-level programming tools

OPERATING SYSTEM: TINY OS

- Tiny OS 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 footprint, Tiny OS chooses to have no file system, supports only static memory allocation, implements a simple task model, and provides minimal device and networking abstractions.

- 
- 
- Tiny OS 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.
 - Tiny OS 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.
 - Tiny OS application example—Field Monitor, 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.

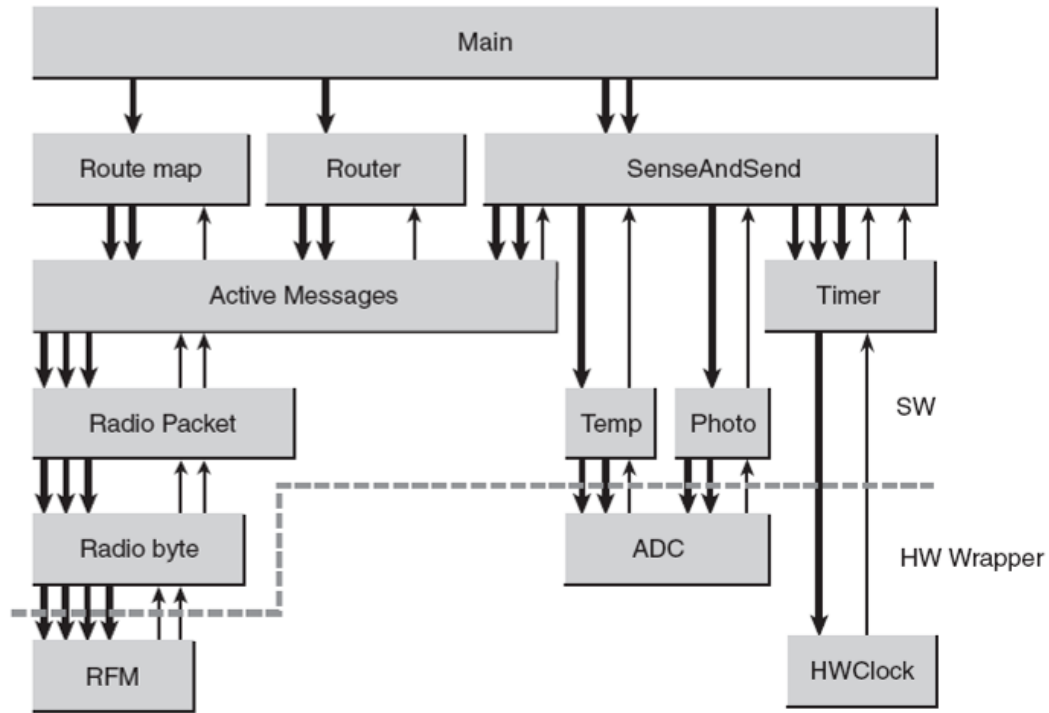
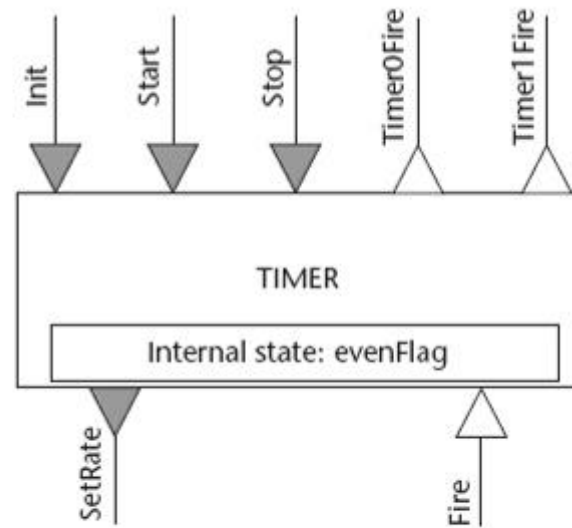




Figure 7.5 The FieldMonitor application for sensing and sending measurements.



THE FIELD MONITOR APPLICATION FOR SENSING AND SENDING MEASUREMENTS



Blocks represent Tiny OS components and arrows represent function calls among them. The directions of the arrows are from callers to callees.





The Timer component and its interfaces.

- 
- 
- 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 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 timer0Fire event to trigger other components; otherwise, it produces a timer1Fire 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 Tiny OS 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 Tiny OS 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.
 - A split-phase execution, sending a packet will block the entire system from reacting to new events for a significant period of time. In the Tiny OS implementation, the `send()` command in the AM component returns immediately.
 - When the packet is indeed sent, the AM component will notify its caller by a `sendDone()` method call.

IMPERATIVE LANGUAGE: NESC

- nesC is an extension of C to support and reflect the design of Tiny OS v1.0 and above.
- It provides a set of language constructs and restrictions to implement Tiny OS components and applications.

COMPONENT INTERFACE

- A component in nesC has an interface specification and an implementation. To reflect the layered structure of Tiny OS, 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 Timer component

The interface definition of the Timer component in nesC.

```
module TimerModule {
  Provides {
    interface StdControl;
    interface Timer01;
  }
  uses interface Clock as Clk;
}
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();
}
```

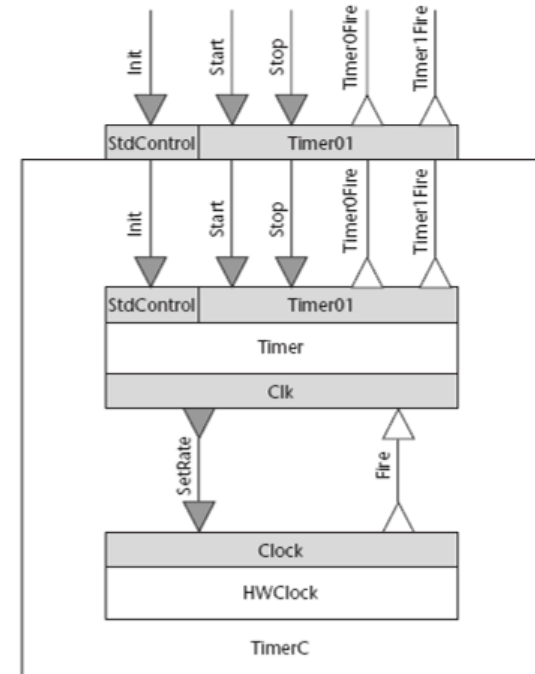
COMPONENT IMPLEMENTATION

- 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.

The Implementation Definition of the Timer Component in nesC

```
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;
  }
  ...
}
```

- nesC also supports the creation of several instances of a component by declaring abstract components with optional parameters. Abstract components are created at compile time in configurations.



The TimerC configuration implemented by connecting Timer with HWClock.

Concurrency and Atomicity



- The language nesc directly reflects the tiny OS execution model through the notion of command and event contexts.
- The implementation definition of the TimerC configuration in nesC.

```
configuration TimerC {
  provides {
    interface StdControl;
    interface Timer01;
  }
}
implementation {
  components TimerModule, Clock;
  StdControl = TimerModule.StdControl;
  Timer = TimerModule.Timer;
  TimerModule.Clk -> HWClock.Clock;
}
```

A section of the implementation of senseandsend, illustrating the handling of concurrency in nesC.



```
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;
}
...
}
```

- 
- 
- 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.



DATAFLOW-STYLE LANGUAGE:TINYGALS

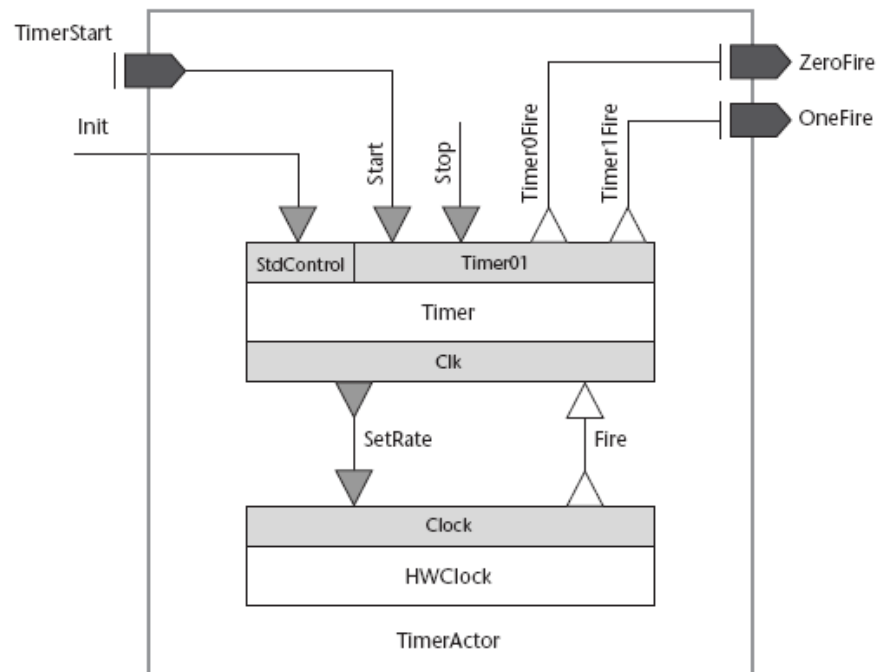
- Dataflow languages are intuitive for expressing computation on interrelated data units by specifying data dependencies among them.
- A dataflow program has a set of processing units called actors. Actors have ports to receive and produce data, and the directional connections among ports are FIFO queues 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.

- 
- 
- 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 Tiny OS.
 - One of the key factors that affects component reusability in embedded software is the component composability, especially concurrent composability.
 - 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 Tiny OS components, including its interfaces and module implementations.
- All method calls in a component 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 Tiny OS components. An actor is constructed by connecting synchronous method calls among Tiny OS components.



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

```
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;
}
```

Implementation of the TimerActor in TinyGALS.]

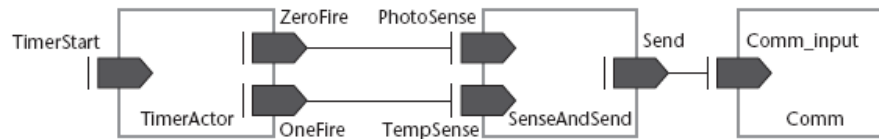





Figure 7.14 Triggering, sensing, and sending actors of the FieldMonitor in TinyGALS.

```

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.

- 
- 
- 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.

- 
- 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 motes. 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.



- 
- 
- 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.



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.



CYCLE-DRIVEN

- 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.
- 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.

$$\begin{aligned}y_k &= f(x_k) \\x_k &= g(y_k)\end{aligned}$$



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 to simulate wireless/mobile networks (e.g., 802.11 MAC and TDMA MAC) and more recently sensor networks.
- SensorSim aims at providing an energy model for sensor nodes and communication, so that power properties can be simulated

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



THE SIMULATOR TOSSIM

- TOSSIM is a dedicated simulator for Tiny OS applications running on one or more Berkeley motes.
- 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 Tiny OS greatly simplifies the design of TOSSIM.

- 
- 
- 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.

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.



- 
- 
- 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:

$$x_{k+1} = f(x_k, u_k)$$

$$y_k = g(x_k, u_k)$$

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.

- 
- 
- 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.
 - The structure of a group defines the “roles” each member plays in the group, and thus the flow of data.

EXAMPLES OF GROUPS

- Combinations of scopes and structures create patterns of groups that may be highly reusable from application to application.
- 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.

N-HOP NEIGHBORHOOD GROUP

- 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.

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.

USING MULTIPLE TYPES OF GROUPS

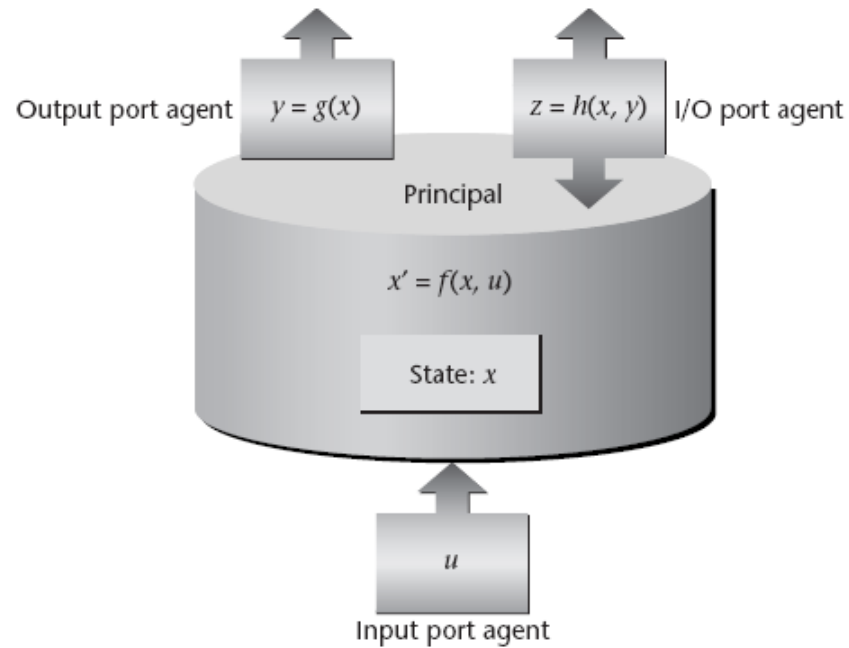
- Mixing and matching groups is a powerful technique for tackling system complexity by making algorithms much more scalable and resource efficient without sacrificing conceptual clarity.
- One may use highly tuned communication protocols for specific groups to reduce latency and energy costs.
- There are various ways to compose groups. They can be composed in parallel to provide different types of input for a single computational entity.

PIECES: A STATE-CENTRIC DESIGN FRAMEWORK

- PIECES (Programming and Interaction Environment for Collaborative Embedded Systems) 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.
 - 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.





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

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.

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.



- 
- 
- 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. 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

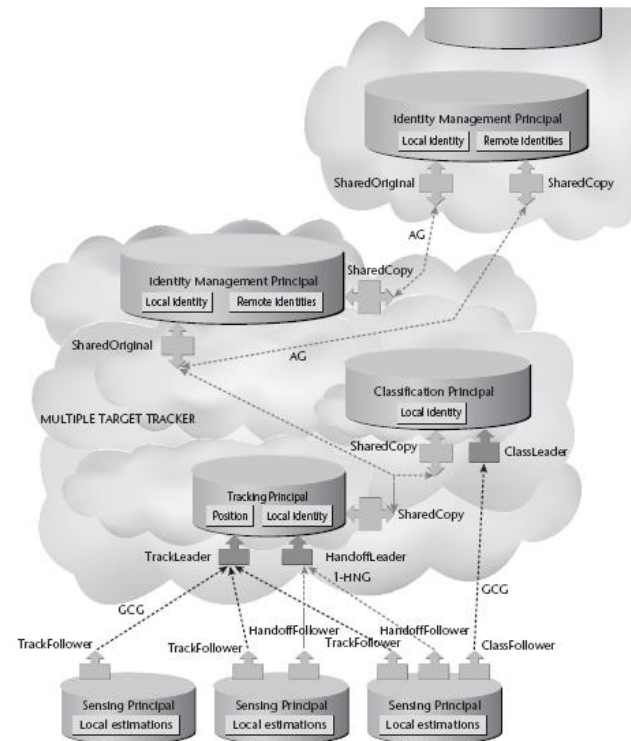
- PIECES provides a mixed-signal simulator that simulates sensor network applications at a high level.
- The simulator is implemented using a combination of Java and Matlab. An event-driven engine is built in Java to simulate network message passing and agent execution at the collaboration-group level.
- 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.

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.

- 
- 
- 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 single-target 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.

- 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.



SIMULATION RESULTS

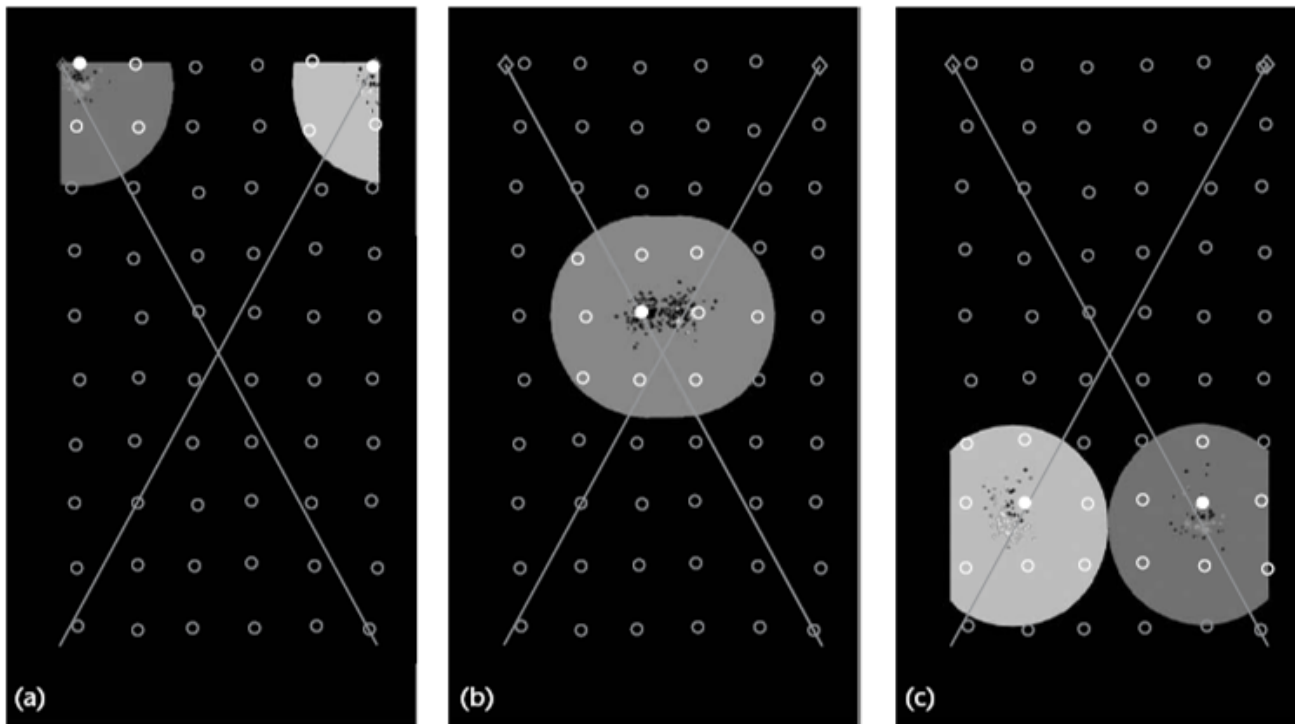


Figure 7.18 Simulation snapshots: Sensor nodes are indicated by small circles, and the crossing lines indicate the true trajectories of the two targets. One geographically constrained group is created for each target. When the two targets cross over, their groups merge into one.



THANK YOU

THE CONTENTS IN THIS E-MATERIAL IS TAKEN FROM THE TEXTBOOKS AND
REFERENCE BOOKS GIVEN IN THE SYLLABUS

