Sengamala Thayaar Educational Trust Women's College
(Affiliated to Bharathidasan University)
(Accredited with 'A' Grade {3.45/4.00} By NAAC)
(An ISO 9001:2015 Certified Institution)
Sundarakkottai, Mannargudi-614 016.
Thiruvarur (Dt.), Tamil Nadu, India

WIRELESS SENSOR NETWORK

P16CS42

A.Srilekha

Assistant Professor

PG & Research Department of Computer Science

S.T.E.T Women's college, Mannargudi.

UNIT 5

SENSOR NETWORK PLATFORMS AND TOOLS

INTRODUCTION

A real-world sensor network application is likely to incorporate all the functionalities like sensing and estimation, networking, infrastructure services, sensor tasking, data storage and query. 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. On the other hand, an application developer must provide end users 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 comprise reactive, concurrent, distributed programs running on ad hoc resource-constrained, unreliable computation and communication platforms. 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.

SENSOR NODE HARDWARE

Sensor node hardware can be grouped into three categories, each of which entails a different trade-off 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), and various personal digital assistants (PDA). These nodes typically run –ff-the-shelf operating systems such as WinCE, Linux, or real-time operating systems and use standard wireless communication protocols such as IEEE 802.11, Bluetooth, Zigbee etc. Because of their relatively higher processing capability, they can accommodate wide variety of sensors, ranging from simple microphones to more sophisticated video cameras.
- Dedicated embedded sensor nodes: Examples include the Berkeley mote family the UCLA Medusa family, Ember nodes and MIT μAMP. These platforms typically use commercial off-the-shelf (COTS) chip sets with emphasis on small formfactor, 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 rarely any operating system support. A classic example is the TinyOS platform and its companion programming language, nesC.

- System on-chip (SoC) nodes: Examples of SoC hardware include smart dust, the BWRC picoradio node [5], and the PASTA node [6]. Designers of these platforms tryto 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. 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.

SENSOR NETWORK PROGRAMMING CHALLENGES

Traditional programming technologies rely on operating systems to provide abstraction for processing, I/O, networking, and user interaction hardware. When applying such a model to programming networked embedded systems, such as sensor networks, the application programmers need to explicitly deal with

message passing, event sync hronization, 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 etc. 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 [8] 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 must 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. There 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 enough. 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 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, then neither the simple client-server model nor the tree organization is optimal. More sophisticated design and methodologies and platforms are required.

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 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.

Operating System: TinyOS

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, 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 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 components specification is independent of the component's implementation. Although most components encapsulate software functionalities, some are just thin wrappers around hardware. An application, typically developed in the nesC language, wires these components together with other application-specific components. 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 pre-empting or being pre-empted by other tasks. Thus, tasks are non-pre-emptive. 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 pre-empts tasks and can be pre-empted by other events. Because there is no pre-emption mechanism among tasks and because events always pre-empt 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. Another trade-off between non-pre-emptive task execution and program reactiveness isthe design of split-phase operations in TinyOS. Like the notion of asynchronous method calls in distributed computing, a split-phase operation separates the initiation of a method call from the return of the call. A call to split-phase operation returns immediately, without performing the body of the operation. The true execution of the operation is scheduled later; when the execution of the body finishes, the operation notifies the original caller through a separate method call. In TinyOS, resource contention is typically handled through explicit rejection of concurrent requests. All split-phase operations return Boolean values indicating whethera request to perform the operation is accepted.

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. 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.

Imperative Language: nesC

nesC is an extension of C to support and reflect the design of TinyOS. It provides a set of language constructs and restrictions to implement TinyOS components and applications. 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 user interface is a set of method calls hiding the lower layer components. Methods in the interfaces can be grouped and named. Although they have the same method call semantics, nesC distinguishes the directions of the interface calls between layers as event calls 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. 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.

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. 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 configuration. As TinyOS does not support dynamic memory allocation, 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 single Std Control interface, which is the ultimate source of initialization of all components.

Dataflow-Style Language: TinyGALS

Dataflow languages are intuitive for expressing computation on interrelated data units by specifying data dependencies among them. A dataflow diagram 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 data flow 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 as language for TinyOS. One of the key factors that affect 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 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. All method calls in a component interface are synchronous method calls- that is, the thread of control enters immediately into the called component from the caller component. An application in TinyGALS is built in two steps: (1) constructing asynchronous actors from synchronous components, 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. 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.

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. 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 theMAC layer or network layer, using, for example, stochastic processes to represent low-level behaviors.

Physical environment model:

A key element of the environment within a sensor network operates is the physical phenomenon of interest. The environment can also be simulated at various levels of details. 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 nodes 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 next tick. This split-phase communication is a key mechanism to reduce cyclic dependencies that may occur in cycle-driven simulations. Most CD simulators do not allow interdependencies within a single tick.

Unlike cycle-driven simulators, a discrete-vent (DE) simulator assumes that the time is continuous, and an event may occur at any time. As event is 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 software module within, and the events can be message passing among these nodes. 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 should not be earlier than that of the input event. Non-causal components require the simulators to be able to roll back in time, and worse, they may not define a deterministic behaviour of a system. 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 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 therefore, 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 fewer complex components and give faster simulations. This is 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. Many of the simulators 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, J-Sim (previously known as JavaSim), and GloMoSim/ Qualnet. The focus of these simulators is on network modelling, protocol 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. Example include TOSSIM [12] for Berkeley motes and Em* for Linux-based nodes such as Sensoria WINS NG platforms.

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. While the original ns-2 only supports logical addresses for each node, the wireless/mobile extension of it 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 two widely known efforts to extend ns-2 for simulating sensor networks: SensorSim form UCLA [15] and the NRL sensor network extension from the Navy Research Laboratory [16]. 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.

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 handler's 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 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 (AOPDV) routing, and temporarily 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 TinyOS 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 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.

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, 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 updates, such as:

$x_k+1= f (x_k,u_k)$

$y_k=g(x_k,u_k)$

where x is the state of a system, u is the system input, y is the output and 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 as clean as represented in the above equations. The relationships among subsystems can be highly complex and dynamic over space and time. The following issues (which are not explicitly tackled in the above equations) must be properly addressed during the design to ensure the correctness and efficiency of the system.

- 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 input 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 ‘non-functional” 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 on top of an intuitive understanding of where and when the operations are performed.

A collaborative group is such an abstraction. A collaborative 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. These are all referred to as agents.

Intuitively, a collaboration groups 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. A software agent that hops among the sensor nodes to track a target is a virtual node, while a real node is physical sensor. Limiting the scope of a group to a subset of the entire space of all agents improves scalability. 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, one can 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.

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. 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. The role of a principal is to update its state from time to time, a computation corresponding to evaluation function f. A principal also accepts other principals' queries of certain views on its own state, a computation corresponding to evaluating function g.

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 agent is also called an observer, sine it computes outputs based on the host principal's state and sends them to their agents. Observers may

be active and passive. An active observer pushes data autonomously to its destination (s0, while a passive observer sends data only when a consumer request for 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 x|s, can only be maintained by exactly one principal. Note that this does not prevent other principals from having local caches of x|s for efficiency and performance reasons; nor does it prevent the other principals from locally updating the values of cached x|s. However, there is only one master copy for x|s. All local updates should be treated as "suggestion" 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.