International Journal of Distributed Sensor Networks, 1: 17–34, 2005
Copyright © Taylor & Francis Inc.
ISSN: 1550-1329 print/1550-1477 online
DOI: 10.1080/15501320490886314

Real-Time Sensor–Actuator Networks

SHIVAKUMAR SASTRY

Department of Electrical and Computer Engineering, 247 Sumner Street, The University of Akron, Akron, OH 44325-3904, USA

S. S. IYENGAR

Department of Computer Science, Louisiana State University, Baton Rouge, LA 70803

Emerging technologies offer new paradigms for computation, control, collaboration, and communication. To realize the full potential of these technologies in industry, defense, and homeland security applications, it is necessary to exploit the real-time distributed computing capabilities of sensor–actuator networks. To reliably design and develop such networks, it is necessary to develop deeper insight into the underlying model for real-time computation and the infrastructure at the node level that supports this model. In this paper, we discuss a new node-level operating system and mechanisms necessary to deploy reliable applications. The overriding issue that guides the design of this operating system is quality of service metric called predictability.

A sensor–actuator network is a distributed platform for integrated computation and control in real-time environments. The nodes in such a network are distinguished by being resource constrained. The power of the network arises from the interactions between simple nodes. Such a network extends the popular distributed sensor networks in several dimensions. After identifying a real-time model, we develop a notion of predictability for a sensor–actuator network. We discuss how the node-level operating system is designed in the resource-constrained environment. An efficient multithreading mechanism and scheduling strategy are required to ensure that local tasks are executed within jitter bounds and that end-to-end delays do not violate application constraints. Mechanisms to support communication, monitoring, safety, fault tolerance, programming, diagnosability, reconfiguration, composability, interoperability, and security are discussed.

Keywords Sensor–actuator network; large-scale distributed system; operating system

1. Introduction

Consider a large-scale system of conveyors that is used to move people, baggage, or material in various engineering applications. The application, such as moving baggage in an airport or moving material in a manufacturing plant, specifies the performance limits and precision within which the system of conveyors must operate. Traditionally, the operations in such a system are regulated using industrial controllers that are connected to a collection of sensors and actuators. These sensors and actuators are connected to the controllers using a variety of wired networks, and sometimes the controllers are interconnected using control networks. Ethernet is the network of choice for such systems, and a large installed base is based on proprietary networks.

When the sensor–actuator network (SANe) discussed in this paper is fully realized (see Fig. 1), the controllers and all the networks connecting various components will be

Address correspondence to Shivakumar Sastry, The University of Akron, Department of Electrical and Computer Engineering, 247 Sumner Street, Akron, OH 44325-3904, USA. E-mail: ssastry@uakron.edu

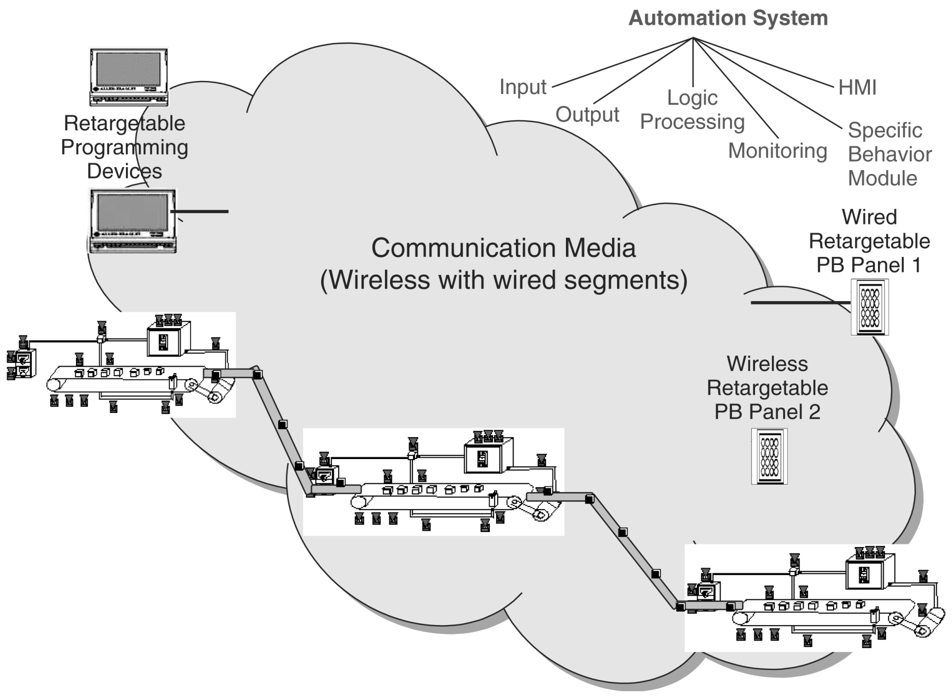

Automation System
Input
Output
Logic Processing
Monitoring
HMI
Specific Behavior Module
Retargetable Programming Devices
Communication Media (Wireless with wired segments)
Wired Retargetable PB Panel 1
Wireless Retargetable PB Panel 2

FIGURE 1 Conception of a SANe.

eliminated. Each sensor and actuator will be endowed with capabilities to perform limited computation and communication tasks. These nodes interact in a peer-to-peer manner to accomplish the objectives of the system, i.e., efficiently move people and materials satisfying application demands.

A SANe is a distributed platform for integrated computation and control in real-time environments [1]. The nodes are distinguished by being resource constrained. Each node comprises one or more sensors and actuators that transduce signals, a transceiver, and a microcontroller that is used for computation and coordination of communication tasks. Simple sensing, actuating, and computing tasks are executed on the microcontroller. Nodes communicate with each other over wired and wireless media. The nodes, and hence, the interactions between nodes, are simple. The power of a SANe arises from the collective operation of the simple nodes.

SANe represents the next generation of distributed systems because of its immense scale, and because its design cannot be based on traditional sequential paradigms such as a file, a document, or an object [2]. A SANe is a representative example of the next generation of real-time systems because of the simplicity of the nodes, and because it must regulate operations in a large-scale, inherently nondeterministic environment using such simple nodes [3, 4, 5]. The SANe extends the popular sensor networks in several dimensions. Figure 2 depicts the emergence of the real-time SANe.

A SANe is a highly engineered system. The nodes are heterogeneous, and the topology is designed to match the demands of the application. SANe nodes must operate within tight performance bounds locally and coordinate with each other to deliver the expected performance at the system level. Low energy consumption is desirable to facilitate scalability, and energy conservation is not a design objective. At a high level, the difference between a sensor network and a SANe is similar to the difference between a desktop

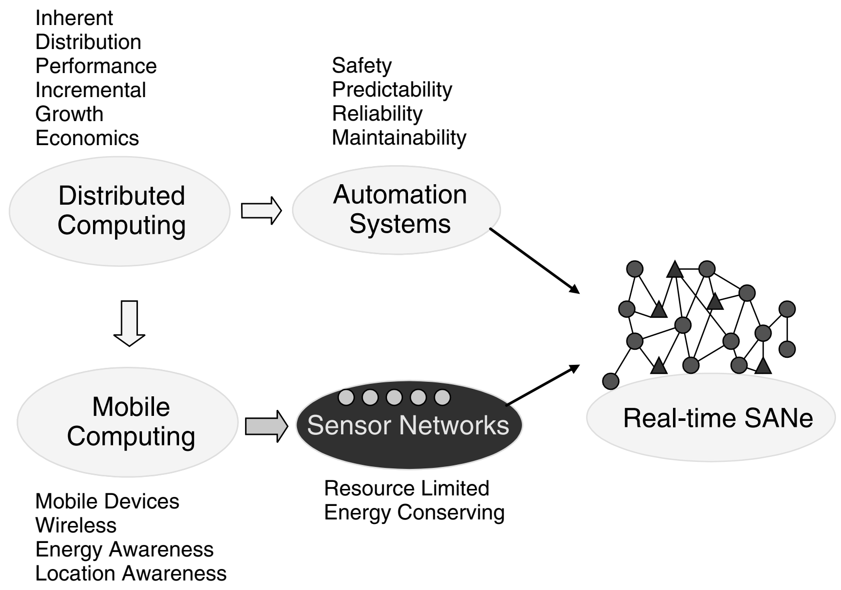


FIGURE 2 Emergence of SANe.

computer and an industrial controller [6]. Instead of considering integration with the physical environment as one of incorporating sensors with special modalities, a SANe is based on a real-time model. Instead of basing the communications on traditional paradigms of stack-based communication, the interaction between nodes in a SANe must be based on higher-level principles such as coordination and robustness. A SANe is also distinguished from general ad-hoc networks and mobile networks, because only a few SANe nodes are mobile, while the location of a large percentage of the nodes is highly engineered.

Engineering applications are distinguished by the following requirements:

- Operate continuously without being reset to a known configuration
- Operate in different modes based on application or user needs
- Support transparent and easy monitoring during execution
- Cope with uncertainties without catastrophic consequences
- Aid users in diagnosing faults and restoring the system to a normal state
- Support simple programming and reconfiguration

Such applications typically involve the coordinated, and periodic, operation of multiple actuators. Different nodes of the SANe operate in different time scales. The actuators cannot be operated in a predetermined manner (e.g., using a programmed finite state machine) because of two sources of uncertainties. First, the operations that must be performed at each step may not be completely known; second, the failures that impact operations at run time may not be predictable. At first sight, a SANe appears to be a new platform for control. Control-related computations account for less than 15% of the total tasks in a SANe. The remaining tasks deal with fault management, information gathering, and monitoring the system at various levels of detail. Thus, the node-level operating system must support efficient execution of such heterogeneous tasks.

Because of the low resources available on SANe nodes and its fundamental principle of being a reactive system, traditional computer science abstractions and the boundaries between these abstractions are blurred. For example, the available data memory precludes the need for memory management. Multiple users cannot work with a SANe node simultaneously, and there is no file system that is available to the node. The microcontroller is typically a low-power 8-bit processor without a deep pipelining structure and no hardware support for executing multiple threads. Thus, the node-level operating system does not address traditional issues that motivate the design of commercial real-time and non-real-time operating systems. The three important problems that must be addressed are as follows:

1. Define a real-time model for a SANe
2. Develop the notion of predictability for SANe
3. Develop node-level operating system (OS) support for achieving predictability

Section 2 presents a real-time model for SANe and relates the different types of nodes to this model. We discuss the nature of tasks and discuss a simple quality of service (QoS) metric for periodic tasks. This section concludes with an example of how a large-scale conveyor system is composed using the ideas discussed. Section 3 discusses the fundamental differences between an event-triggered architecture and a time-triggered architecture for a SANe node. Section 4 presents a notion of predictability for SANe based on two concepts, namely, predictable actuation and predictable tracking. Section 5 discusses the critical issues in the design of a new node-level operating system that can achieve the predictability for a SANe. After discussing related work in Sec. 6, we conclude this paper in Sec. 7.

2. Sensor–Actuator Network

The underlying real-time model for a SANe guides the development of the notion of predictability. Our starting point is a basic definition of a SANe as a graph, $SANe = (V, E)$, where V is a set of nodes, and E is a set of edges. Each edge, $e \in E$, represents a possible communication link between two nodes, i.e., $e = (u, v)$, $u, v \in V$. We discuss the nature of these nodes and edges and the nature of the tasks executing on a SANe in this section.

The abstract SANe defined above manifests physically in a network of embedded microcontrollers. One or more SANe nodes are hosted on a microcontroller that is capable of wireless or wired communication. Depending on the application needs for safety and fault tolerance, a SANe node may be replicated and simultaneously hosted on more than one microcontroller. Thus, some interactions between SANe nodes are internal to a microcontroller, while other interactions are between microcontrollers.

2.1 Real-Time Model

To obtain a real-time model for a SANe, we extend a traditional real-time model discussed by Koeptz [7]. We use the idea of RT-Entity as defined by Koeptz.

Definition 1. An *RT-Entity* is an element of interest in the system. An RT-Entity that occurs in the engineering application is called an *external RT-Entity*. Similarly, an RT-Entity that occurs within the computing and communication infrastructure (e.g., microcontrollers) is called an *internal RT-Entity*.

A bag arriving at a conveyor segment and a product falling off the conveyor system by accident are two examples of external RT-Entities. An interrupt occuring in a microcontroller and completion of a communication or scheduling task are examples of internal RT-Entities. A mode change command initiated by an operator is an external RT-Entity. In response to this command, the node at the interface between the SANe and the operator generates internal RT-Entities.

Each RT-Entity occurs in the neighborhood of a microcontroller. Note that both external RT-Entities and internal RT-Entities can occur in the neighborhood. We use an intuitive notion of neighborhood that encompasses both physical connectivity and the (x, y, z) coordinate of the microcontroller in a standard three-dimensional (3-D) space. We utilize the idea of observation as defined by Koeptz with a small modification.

Definition 2. An observation, O, of an RT-Entity e is a triple $O = <$ *name, value, location* $>$, where *name* is the identity of the RT-Entity; *value*, v, is an estimation of the state of the RT-Entity; and *location* is the (x, y, z) coordinate of the microcontroller in which O occurred.

Definition 3. An observation is said to occur when a microcontroller records the state of an internal RT-Entity or estimates the state of an external RT-Entity.

These definitions lead to the recognition that an observation may occur in one microcontroller and be used in more than one microcontroller of the network. We assume a publisher–subscriber model for the RT-Entities. Because of the multihop communication in the SANe, there is a temporal lag between when the RT-Entity occurs and when it is used. Once again, we use the idea of temporal accuracy, defined by Koeptz, to ascribe a quality to the validity of an observation. However, we clarify this idea and note that the temporal accuracy is not associated with an observation. It is associated with the use of an observation. Naturally, if the temporal accuracy of any use of an observation violates application constraints, the SANe will not meet the specified objectives.

At first sight, a SANe appears to be a trivial application of the current sensor networks technology. This view is simplistic and not true. SANe creates a fundamental paradigm shift in the relationship between a controller and controlled application, as depicted in Fig. 3. Part (a) of Fig. 3 shows the traditional model in which the controller is considered to be a separate entity that regulates the operations in an application using the sensors and actuators. In contrast, advances in distributed sensor networks technologies have made it possible to realize models of interaction that are shown in part (b) of Fig. 3. Both views shown in part (b) are simultaneously true, particularly when the computational and sensor–actuator technologies are deeply embedded in applications.

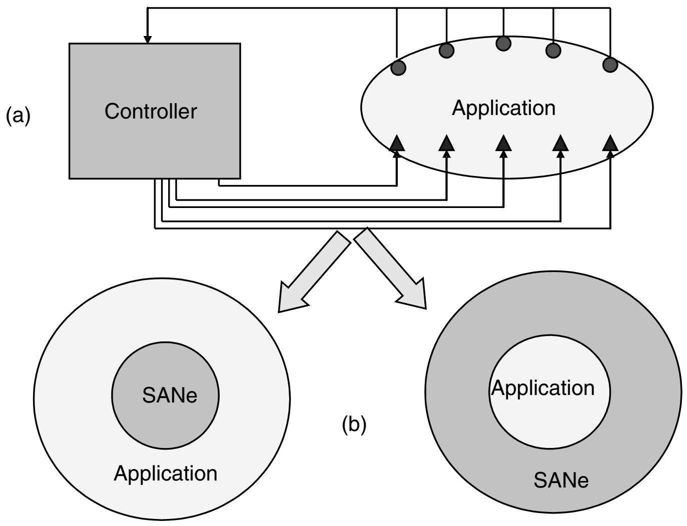


FIGURE 3 Fundamental paradigm shift.

To obtain predictable behavior from the SANe, all nodes must either act on the same observation at the same time or cope with the temporal accuracy of an observation.

RT-Entities provide a window into the dynamics of the engineering application and the SANe. One of the unique features of the next-generation control and computation frameworks such as SANe, is that the traditional distinction between the controlled application and controller that pervades classical control theory is now blurred. In a SANe, the RT-Entities are specialized as different types of nodes, each with unique functional characteristics.

2.2 Types of Nodes

The nodes in a SANe can be partitioned into the following six sets:

$$V = S \cup A \cup T \cup C \cup B \cup H$$

where $S = \{s_1, s_2, \cdots, s_{n1}\}$ is a set of sensor; $A = \{a_1, a_2, \cdots, a_{n2}\}$ is a set of actuators; $T = \{t_1, t_2, \cdots, t_{n3}\}$ is a set of timers; $C = \{c_1, c_2, \cdots, c_{n4}\}$ is a set of counters; $B = \{b_1, b_2, \cdots, b_{n5}\}$ is a set of stored observations; and $H = \{h_1, h_2, \cdots, h_{n6}\}$ is a set of human interaction devices. For clarity, we refer to each partition as a functional-element.

2.2.1 Sensors. Every sensor, $s_i \in S$, estimates the state of an external RT-Entity. The fidelity of the estimation depends on the modality of the sensor, the effectiveness of the transducing device, and the rate at which the estimate is periodically generated. The objective of each sensor is to estimate the state of an external RT-Entity and facilitate the microcontroller to record the observation.

Although it appears that the sensor node is a simple input device (especially for a discrete sensor), in reality, it is a little more complex. There is a default state of each sensor s_i. For example, a discrete sensor may be *Normally Open*, denoted $\ddot{s}_i$, and detect the presence of a signal, or it may be *Normally Closed*, denoted $\ddot{s}_i$, and detect the absence of a signal. If the sensor has additional electronics and physics support, it may revert to known values when it detects a local failure in the sensing activity, or it may provide additional confirmation data that help in estimating the level of confidence in the observation.

Because a SANe is a periodic system, the estimate of an external RT-Entity is only valid for the duration of the periodic scan rate. If the temporal accuracy of any use of an observation exceeds the periodic scan rate, the node using the observation must account for the inaccuracy. The value of an RT-Entity in a node is overwritten when a new value has been obtained at the location of its occurrence. The value of an observation in the SANe, at all locations $(x_i, y_i, z_i) \neq (x, y, z)$, is overwritten only when a new observation is received over an edge.

2.2.2 Actuators. Every actuator, $a_i \in A$, initiates an external RT-Entity. The a_i maps the value of an internal RT-Entity, e_i, to an appropriate action that initiates an external RT-Entity. For example, a Boolean value (true or false) may be used to initiate one or more physical (electronic or mechanical) actions in the environment, such as open or close an actuator or rotate a motor. The actuator, a_i, is responsible for computing a value for e_i.

Actuators that require the signal to be asserted high to initiate operations are represented as $\ddot{a}_i$ and actuators that require the signal to be asserted low are represented as $\overline{a}_i$. Actuators that require a pulse input are represented as $\hat{a}_i$. Noting that the SANe is a periodic system, the effect of an actuation persists only as long as the computed value of e_i does not change. For pulsed actuators, we assume that the actuation effect is atomic. The SANe uses the latest value for each observation that is necessary to compute e_i.

2.2.3 Timers. Every timer, $t_i \in T$, is an internal RT-Entity. The function of a timer is to provide timing signals to SANe nodes at specified durations. Each such signal is referred to as a *tick*. The variable t_i^{Acr} specifies the duration of time between two successive ticks of t_i. The t_i^{Val} specifies the duration of time for which the timer remains ticking.

Every timer accepts a Boolean signal t_i^{Res} and restarts its timing operation in response to this signal. Two other Boolean signals, t_i^{Start} and t_i^{Stop}, start and stop t_i. When t_i^{Stop} is asserted, the timer aborts its current timing operation. There are two Boolean values $t_i^{Ticking}$ and t_i^{Done} that represent the dynamic state t_i. When t_i is maintaining time (ticking) normally, the signal $t_i^{Ticking}$ is asserted high. When a duration specified in t_i^{Val} has elapsed, t_i^{Done} is asserted high and remains high until t_i is reset. When t_i^{Done} is asserted high by the timer, it asserts $t_i^{Ticking}$ to low, and vice versa.

Any number of timers can be defined in a SANe. Typically, each timer relies on the hardware timers of the microcontroller to maintain accuracy. The value of a timer is available as an observation to SANe nodes.

2.2.4 Counters. Like timers, every counter, $c_i \in C$, is an internal RT-Entity. The function of a counter is to monitor the number of times that an event of interest occurs. c_i^{Val} specifies the maximum value that is expected to be reached in c_i. We assume that the range of numbers that can be represented in a counter is $0,1,2,\cdots,c_i^{Val}$, and that every counter is of arbitrary precision. When the maximum value of a counter c_i is attained, a value c_i^{Done} is asserted high.

There are three Boolean signals associated with every counter. c_i^{Res} causes the value of the counter to be set to a predefined value (either 0 or some other value if excess-*k* notation is being used to represent negative numbers). c_i^{Up} and c_i^{Done} cause the value stored in the counter to either increment or decrement by 1. Any number of counters can be defined in a SANe. The value of a counter is available as an observation in a SANe.

2.2.5 Stored Observations. Every stored observation, $b_i \in B$, is a copy of an observation in the SANe.

2.2.6 Human–Computer Interaction Devices. A human–Computer interaction (HCI) device, $h_i \in H$, is bipartite collection of RT-Entities as shown in Fig. 4. There are an equal number of external RT-Entities and internal RT-Entities. Every external RT-Entity is paired with a unique internal RT-Entity. The device h_i ensures that the values of the external RT-Entities and the internal RT-Entities are the same. HCI devices are used to monitor data from SANe and to specify the execution behavior of one or more nodes. The external RT-Entities are further partitioned into prompts and commands. The SANe generates the prompts to guide the actions of operators. Operators issue commands to direct the behavior of one or more nodes of the SANe.

2.3 Tasks in a SANe

In each microcontroller that hosts one or more SANe nodes, there is a set of tasks $T = \{\tau_1, \tau_2, \cdots, \tau_n$ that are executed periodically. Each τ_i is a collection of instructions, and we say that a task is executed when all the instructions are executed in the microcontroller. The time to execute a task is called the computation time, $\chi(\tau_i)$, of τ_i. Note that tasks are associated with a microcontroller that hosis SANe node(s).

Computation tasks, τ_i^c, execute by utilizing the resources in a single microcontroller. In contrast, a communication task, τ_i^x, transfers data between a source and a destination. Thus, $\chi(\tau_i^x)$ represents the sum of the times required for executing instructions on the

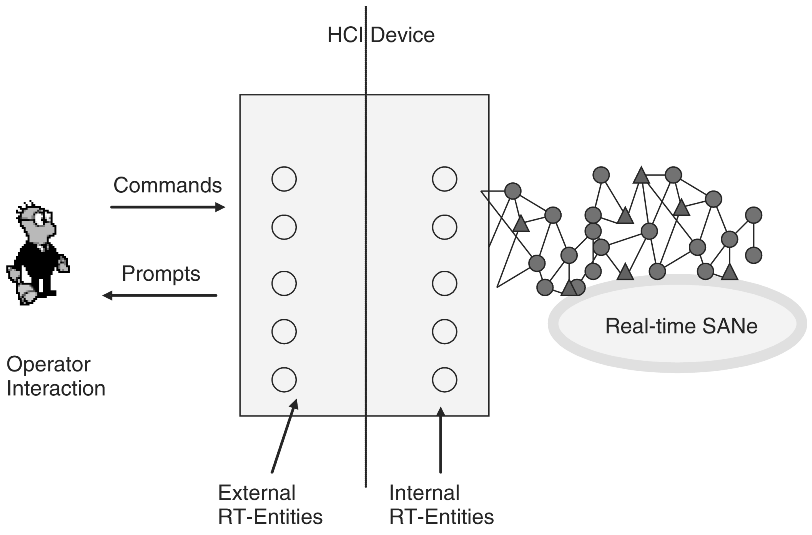


FIGURE 4 RT-Entities in an HCI device.

source microcontroller to initiate a message, propagating the message over a communication media, and executing instructions on the destination microcontroller to receive the message. Whenever a message must be routed through intermediate microcontrollers, we measure the communication time by adding the time for each hop.

2.4 A Simple Metric for QoS

As a first step, one may assume that every task, τ_i, is performed periodically at a rate $R(\tau_i)$. Ideally, τ_i would repeat at a specified rate accurately. In practice, because of changes in the operating environment and because of approximations that are inherent in discretized systems, the actual rate of execution of a periodic task is $R(\tau_i) \pm \varsigma(\tau_i)$. $\varsigma(\tau_i)$ is called the jitter associated with task τ_i.

The jitter associated with a task varies with the manner in which the task is implemented. For example, if the task is carried out using hardware devices, the jitter tends to be low. In contrast, if a task is carried out via software, the jitter tends to be higher. The jitter associated with a fixed sequence of instructions is lower than the jitter associated with computations involving many decision points. The jitter associated with communication tends to be large when the nodes are responsible for detecting and avoiding collisions in the media. Because of uncertainty encountered during signal propagation, jitter encountered when communicating over wireless media tends to be larger than that for wired media.

Such an idea of tasks being periodic leads us to the following simple QoS metric.

Definition 4. Predictability of a Periodic Task. Ideally, a task is said to be predictable when $\varsigma(\tau_i) \to 0$. In practice, for a fixed ε, a task is said to be predictable if $\varsigma(\tau_i) \leq \varepsilon$.

This simple notion of predictability could be extended to the SANe if all tasks were periodic.

2.5 Variety of Tasks

The set of tasks T executing in a SANe have different characteristics. Some tasks are periodic, while others are sporadic. The granularity of some tasks, approximated by $\chi(\tau_i)$

is low, while the granularity of other tasks is high. Some tasks are critical, and others are not critical. Some tasks are hard real-time tasks, while others are soft real-time tasks. Some tasks have higher priorities than others, and these priorities may change with the mode in which the SANe is operating. Such considerations make it difficult to develop a precise notion of predictability of a SANe.

Further, tasks are invoked by a node-level task scheduler. The scheduler is designed to comply with a system-wide scheduling policy and must consider the priority, the deadline, and the latency of a task before scheduling it for execution. Thus, the effects created by the task scheduler must be accommodated in the notion of predictability.

Before developing the notion of predictability further, we first examine the dynamic behavior of a SANe.

2.6 Typical Execution in a SANe

Sensors periodically estimate the state of external RT-Entities and create corresponding internal RT-Entities. These internal RT-Entities are transmitted to other microcontrollers as observations. The observations are used by actuators to periodically compute values that are used to initiate other external RT-Entities. Every actuation is based on a set of observations, and these values are periodically updated.

A SANe is required to operate in several modes. For example, in the automatic mode, it must perform periodic operations with minimal operator interaction; in the *k*-hour run mode, which is typically used in manufacturing environments, the SANe must perform all the required actions without affecting any, and in the manual mode, it guides users to interact with the actuators in a safe manner.

2.7 SANe for Conveyor Systems

We describe a complete conveyor system application using the definitions in this section.

Definition 5. A conveyor system, $CS = (G, X, L, O)$ is a 4-tuple, where

- G is a set of segments.
- X is a set of transfer points.
- L is a set of crossover slides.
- O is a set of turnaround modules.

Each segment, $g_i \in G$, is an artifact on which material or people (entities) move. The segment has two ends that we refer to as the *upstream-end* and the *downstream-end*. Each segment is associated with an actuator g_i^a, and we assume that the directions of traversal of each segment are consistent. Every segment has two sensors: one is attached to the upstream-end, and the other is attached to the downstream-end. We denote these sensors as $g_i^{s_u}$ and $g_i^{s_d}$, respectively.

The set of transfer points, $X = \{x_1, x_2, \cdots, c_m\}$, is a collection of virtual points that occur at the intersection of two segments. At every transfer point, the downstream-end of one segment is juxtaposed with the upstream-end of the following segment. Transfer points represent the locations at which an entity moves from one segment to another.

The set of crossover slides, $L = \{l_1, l_2, \cdots, l_p\}$, is optionally placed in the conveyor system. At each crossover slide, entities can be optionally transfered from one segment to another. A crossover slide cannot be located at a transfer point. Every crossover slide is unidirectional and comprises two ends: an upstream-end and a downstream-end. Both ends of every crossover slide are connected to different segments. Entities are only transferred from the segment connected to the upstream-end to the segment connected to the downstream-end of the crossover slide.

The set of turnaround modules, $O = \{o_1, o_2, \cdots, \phi_q\}$, is also placed optionally in the conveyor system. Each turnaround module contains four ports, namely, *A, B, C*, and *D*. Port *B* of every turnaround module must be juxtaposed with the downstream-end of some segment, and port *C* must be juxtaposed with a upstream-end of some segment. Port *A* may be optionally juxtaposed with the downstream-end of some segment, and port *D* may be optionally juxtaposed with the upstream-end of some segment. Entities that arrive via ports *A* and *B* will be transferred to a segment juxtaposed at port *C*, if no segment is juxtaposed with port *D*. If segments are juxtaposed with both ports *C* and *D*, then for every entity that arrives on the turnaround model, there exists a specification to direct the entity to a port.

Using segments, turnaround modules, and crossover slides, we can construct arbitrarily large conveyor systems. A SANe for the conveyor system is constructed by associating a microcontroller with each segment of *CS*. SANe nodes that correspond to sensors, actuators, timers, and counters are hosted on the microcontroller.

Ignoring the difference in interactions between segments and turnaround modules, and between segments and crossover slides, the following dominant-operations must be performed by the SANe at each segment g_i:

- When the downstream sensor $g_{i-1}^{S_d}$ detects the presence of a part, g_i^a is initiated.
- When the upstream sensor $g_i^{S_u}$ detects that the part has moved across the transfer point, the downstream segment, g_{i-1}^a, is turned off to conserve power.
- When a segment initiates its actuator, it starts a watchdog timer.
- When the actuator of a segment is turned off, its watchdog timer is also turned off.
- If the watchdog timer is done and the actuator is still on, g_i^a is turned off.

There is a collection of monitoring stations that can be located anywhere in the vicinity of the conveyor system. By defining the span of control associated with each HCI device, stored observations and application-level control tasks, such as mode change, monitoring, diagnostics, fault management, etc., can be suitably defined.

Figure 5 shows a conveyor system that comprises 21 segments, four turnaround modules, and two crossover slides.

The dynamics of the SANe and interactions between various RT-Entities are governed by a fundamental architectural decision that is discussed next.

3. Event-Triggered Versus Time-Triggered Architectures

The choice between an event-triggered architecture and a time-triggered architecture for a SANe is critical, because it is a decision that affects the dynamics of the application and the SANe. For reasons explained in this section, we believe that the node-level operations and critical internode interactions must be time triggered. Noncritical interactions can be event triggered.

To implement an effective event-triggered architecture such as TinyOS [8], the microcontrollers must be able to cope with event showers that can arise. Typical industrial automation applications have the characteristic that in the normal mode, the communication traffic is low and localized, while under abnormal circumstances, the traffic is bursty and pervades over a large number of nodes. Explicit flow control is necessary to regulate communications. These two requirements increase the demands on the node-level resources. Because only the source microcontrollers have knowledge of the RT-Entities, the communication protocols must provide support for fault management, clock synchronization, security, programming, etc. It is also not easy to implement replica determinism in an event-triggered architecture without considerable computing and communication

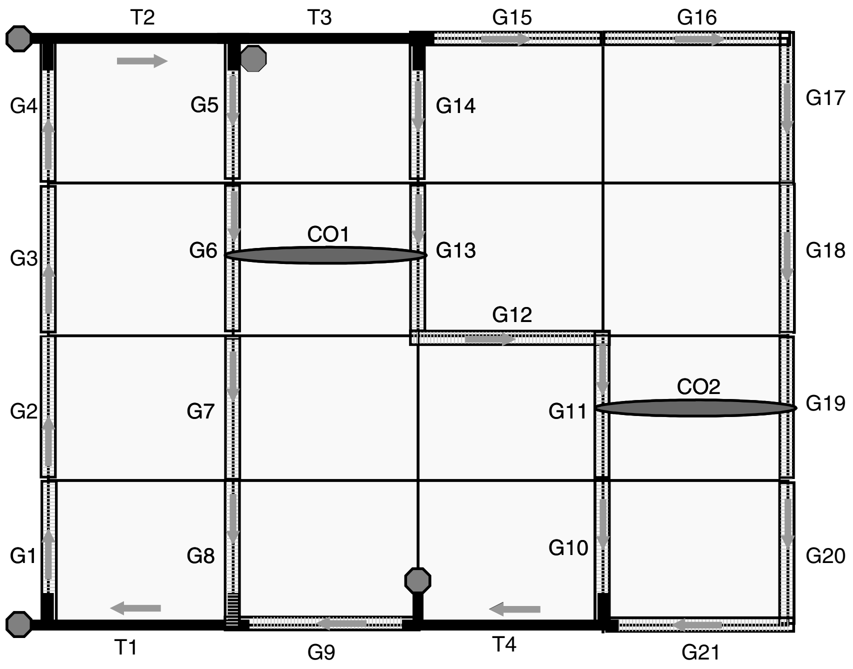


FIGURE 5 An example conveyor system.

overheads [7]. Therefore, despite the appeal of easy configuration and adaptability, event-triggered architectures are unlikely to succeed in large-scale SANe (over 200,000 nodes) of the future.

In contrast to the event-triggered architectures, the time-triggered approaches are inherently predictable. There are no event showers to contend with, and flow control is implicit. The periodic foundation can be used to implement clock synchronization and mechanisms to cope with the temporal accuracy of observations. We can add microcontrollers to the SANe without redesigning the protocol, and it is relatively easy to implement fault tolerance. One often-cited criticism is that such architectures tend to be inflexible and overdesigned. We believe that this criticism can be adequately addressed by basing the design on theories of coordination of large-scale distributed systems that are likely to be developed [9].

4. Predictability of a SANe

Our view of the predictability of a SANe is consistent with the views of Le Lann [10] and Stankovic and Ramamritham [3]. We believe that the future events in a SANe cannot be predicted with absolute certainty, and that the infrastructure must be designed to cope with uncertainties that may come to pass during execution. To develop a notion of predictability, we first explore the characteristics of a SANe along the five dimensions suggested in [3].

SANe is a reactive, real-time system. There are two kinds of tasks that execute at each node. We refer to actions, such as sensing, actuating, and simple computations that are directly related to computing the next values for actuators, as *Control tasks*. We refer to all other tasks in the nodes, including communication, monitoring, fault management, security, programming and other such tasks, as *Systems tasks*.

Control tasks have small granularity. The latency associated with these tasks is typically low, and they are scheduled for execution as late as possible by the scheduler. The granularity of systems tasks are larger. The latency will depend on the mode in which the SANe is currently operating. This mode also dictates the criticality, the priority, the deadline, and the strictness of the deadline for the tasks.

The notion of predictability that we develop for a SANe is based on two ideas: the predictability of actuations and the predictability of tracking.

4.1 Predictable Actuation

Consider an actuator $a_i \in A$. A value for internal RT-Entity of a_i, e_i, is computed as a function of one or more observations of the SANe. The microcontroller that hosts a_i is responsible for computing e_i. a_i uses observations of RT-Entities that possibly occur in the neighborhood of some microcontroller. These RT-Entities correspond to the different types of nodes of the SANe. Each observation involved in the computation has a temporal accuracy. Our idea of predictable actuation is based on nothing that the temporal accuracy of all such observations must not violate application constraints. Both control tasks and systems tasks are involved in achieving predictable actuation.

Definition 6. Predictable Actuation. We say that an actuation in a SANe, i.e., a periodic computation of a value for an internal RT-Entity and a mapping of this value to the corresponding RT-Entity, is predictable whenever the temporal accuracy of every observation involved in computing the value of e_i is within the demands of the application.

4.2 Predictable Tracking

Transparent monitoring is one of the important features of a SANe. Thus, we believe that predictability in monitoring and tracking operations in a SANe is critical. We assume that all SANe nodes have access to a global time base and are operating with synchronized clocks.

There are two notions of tracking in a SANe. First, when an observation is presented to an operator via some HCI device, we say that the behavior of the SANe is being tracked by operators. Second, when an operator issues a command via some HCI device, and this command results in computation of the value for some actuators, $\{a_i\} \subset A$, we say that the actuators are tracking the command. Thus, for every instance of tracking in a SANe, there are two end points.

Definition 7. Predictable Tracking. We say that the SANe is being predictably tracked whenever messages are delivered between the end points in the same temporal order in which they were generated.

4.3 A QoS Metric for a SANe

We assume that the SANe operates in one of two modes: fully automatic or fully manual. In the fully automatic mode, the actuations are performed periodically, and there is minimal user interaction with operators. In the fully manual mode, the actuations are performed only when a command (which is an external RT-Entity in some HCI device, $h_i \in H$ is issued by an operator. The SANe is expected to supply the prompts that are necessary to ensure that no command issued by an operator can lead to catastrophic damage of the SANe or the engineering application.

Definition 8. We say that a SANe is predictable in the fully automatic mode when all its actuations and tracking are predictable.

Definition 9. We say that a SANe is predictable in the fully manual mode when all its tracking is predictable.

5. OS Design Issues

A new node-level operating system is an important component that helps in achieving predictability of the SANe discussed in the preceding section. In Sec. 6, we discuss the differences between this node-level operating system and existing operating systems in detail. In the remainder of this section, we discuss issues related to the following:

- Multithreading
- Synchronization, mode change, and communication
- Communication protocol
- Scheduling
- Integrated mechanisms for security, programming, monitoring, and fault management

5.1 Efficient Multithreading

Control tasks and system tasks executing in a SANe node are well defined a priori. Because of the resource-constrained nature of SANe nodes, these threads must be implemented without consuming excessive node resources [11]. In addition, we note that dynamic memory allocation and deallocation introduce overhead and nondeterminism at the node level. The data memory of the microcontroller must be preallocated, and all the executing tasks must be resident in the program memory. Each executing task in a SANe node is associated with an operating mode and a mode-dependent priority. The effort in switching tasks, e.g., register copies and state restoration, must be minimized by a mode-aware scheduler.

5.2 Scheduling

There are two fundamental approaches to scheduling: fixed priority scheduling and dynamic priority scheduling [12, 13]. Because the SANe operates in an environment that is inherently nondeterministic, it is necessary to change the priorities of tasks executing in the nodes to cope with changes. However, because of the limited computational resources, we cannot implement a dynamic scheduler in every node. Such limited resources also deter the implementation of rate-monotonic scheduling strategies. Further, the critical need in a SANe is to ensure that the end-to-end delays for actuations and tracking satisfy application constraints. A new mode-aware scheduler that ensures that the actuations are predictable must be developed.

5.3 Synchronization, Mode Change, and Communication

The most critical tasks in a SANe that must be executed at high priorities in all operating modes are related to synchronization, mode change, and communication. Of these tasks, the communication tasks are executed at the highest priority. The nodes distinguish between communications that occur during normal operations and abnormal operations. We are able to reasonably predict the communications requirements during normal operations. We need simple distributed implementations for clock synchronization [2] and protocols for changing modes. In the literature [14], mode change is considered in the general setting of a change in the task pool and priorities that must be executed. In a SANe, we must consider distributed algorithms that can propagate the mode-change commands, we must ensure that the nodes do not violate safety requirements, and finally, we must determine the infrastructure that is necessary in the node-level operating system to support such mode changes.

5.4 Communication Protocol

Traditional approaches based on protocol stacks such as OSI or TCP will not be compatible with the resource-constrained nodes of a SANe. The communications will be supported with a simple three-layer stack comprising the physical layer, a data-link layer, and a network layer. To reduce protocol overheads and perform within the temporal constraints, we believe that a new communications mechanism must be developed to integrate with the time-triggered node architecture of SANe nodes.

5.5 Integrated Mechanisms

To reliably operate engineering applications, a SANe must include support for security, programming, monitoring, and fault management in the node-level operating system.

Because the SANe must adopt to a changing operating environment, wireless connectivity between nodes is desirable. However, the inherent open nature of the wireless media raises many security issues. It is neither feasible nor necessary to execute traditional cryptography algorithms on the values of RT-Entities. Because a SANe is typically installed in an environment that is already protected by an auxiliary perimeter security infrastructure, it is necessary to develop distributed implementations of security mechanisms that operate in the nodes that are close to HCI devices.

To support the two notions of tracking that were discussed in Sec. 4, we first need an effective method to exchange messages. The communication infrastructure must generate time stamps for each originating message and synchronize the clocks of the nodes as the message traverses the SANe. We need additional nodes that implement data-fusion algorithms and strategies to aggregate messages and appropriately limit the communication traffic [15].

The fault-management activities in a SANe include detection, isolation, reporting, and recovery of faults. The primary consideration before executing any such tasks is to ensure that the environment in the neighborhood of the faulty node is safe. This means that the default state of actuators must be set to known values when a fault is detected. Further, the fault detection and reporting strategy must conform to the overall fault-management philosophy of the SANe. When a fault is detected in a node, the neighboring nodes must collaborate to report such faults and aid in recovering from the faults.

5.6 Programming Issue

The programming problem is challenging. The application needs are to support both off-line programming and online programming. Off-line programming is used to specify the desired behaviors when the SANe is not operational. Online programming is used to modify the behaviors in a node when the node is executing in one of the specified modes. Implementing support for such programming in the resource-constrained environment will be a subject of intense research in the coming years. We believe that a node-level programming model will not suffice, and a broader perspective such as the IEC-61499 standard is essential [16]. We believe that approaches that attempt to extend the standard desktop application programming models to SANe nodes will meet with limited success. If we must maintain the simplicity of the nodes, it is important to find a reliable and simple mechanism to alter the node behavior without the overhead of a large operating system.

5.7 Architectural Issues

A SANe comprises a loosely coupled collection of nodes that are deeply embedded into the application. It is beneficial to leverage any organizational structures of the application to identify collections of SANe nodes that must cooperate to deliver subsystem objectives. Traditional automation systems have used hierarchical organizations to deal with scale. Such organizations have sometimes been compatible with system structures. In other cases, the hierarchical organizations have introduced serious addressing and configuration problems. Because SANe is a large-scale system, one critical architecture issue is to provide support at the operating system level to compose loosely coupled teams of nodes that are formed along the structures in the application. Such structures are useful in propagating modes and alarms.

6. Related Work

Many aspects of wireless sensor networks are discussed in the literature [17]. While issues related to integration with physical environments have been considered [18], the underlying real-time model distinguishes a SANe from general wireless sensor networks. The issue of time synchronization has only been addressed briefly in [19, 20]. Sensor networks are typically used in data gathering and monitoring applications. The topology of the network is ad hoc, and the research emphasis is primarily on networking issues that tie the nodes together. Energy-efficient operation is one of the primary design objectives.

6.1 Relation to Existing OS

One major difference between the operating system proposed here and commercial operating systems (general-purpose or real-time) is that we do not focus on mechanisms for device abstractions. In addition, we do not consider the issue of programming support or an application programming model.

Because of the widespread use of embedded systems in diverse applications, there has been a great interest in understanding how an operating system can be customized for various applications [21, 22]. Other methods including TinyOS [8], approach the issue of designing a flexible operating system by using components. Energy efficiency has also been considered [23]. Issues related to adaptation initiated by humans or the application at the user level and the kernel level are discussed in [22].

6.2 Relation to TinyOS

The new node-level operating system discussed here is fundamentally different from TinyOS because of its real-time model. The component layering (by snapping together pieces) and the structure of components, i.e., fixed-size frame, command and event handlers, and bunch of tasks, appear to comprise a useful model for certain systems that can be based on an event-triggered architecture. As depicted in Fig. 6, commands and events are two important entities.

The event model and multithreading, with two-level scheduling, architecture for TinyOS is attractive and appears to fit well in a concurrency-intensive environment. However, it supports no mechanisms to improve determinism and control jitter. In fact, because of the multithreading architecture, all node-level jitter is presented at the communications interface, because there are no buffers. As discussed in Sec. 3, we need explicit mechanisms to implement flow control and manage event showers that are likely to occur whenever abnormal operating situations are encountered.

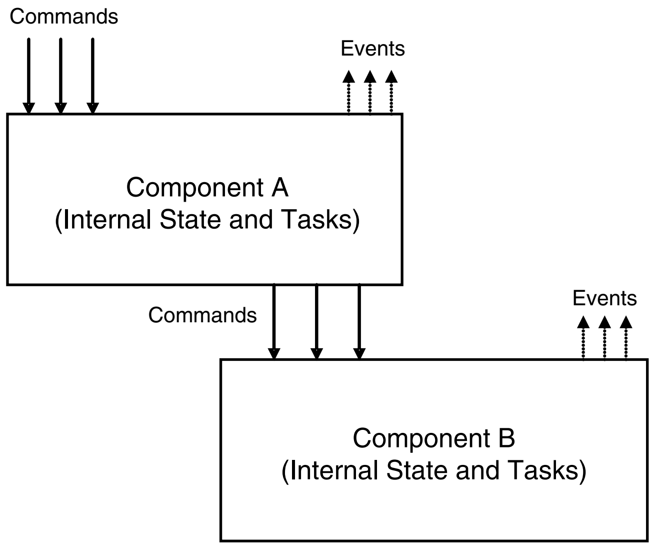


FIGURE 6 Commands and events in TinyOS.

Power conservation is an important design objective for TinyOS. TinyOS makes optimum use of resources in the setting of a low-power, resource-constrained node. While limiting power consumption is important in SANe nodes, conservation is not a design objective.

7. Conclusions

SANe represents a challenging and new class of distributed real-time systems. This paper presented a real-time model for SANe, a definition of predictability, and the design issues involved in achieving predictability. A SANe is said to be predictable if all the actuations are predictable and if the tracking messages are delivered in the same order as they are generated.

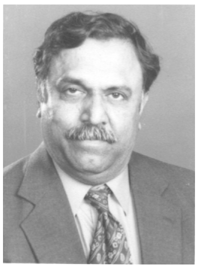
A new node-level operating system must support an effective mechanism for multithreading and a mode-aware scheduler. Three critical tasks that must be executed at a high priority are communication, synchronization, and mode change. The operating system must include mechanisms to support security, programming, fault management, and monitoring tasks.

References

1. J. Agre, L. Clare, and S. Sastry, “A taxonomy for distributed real-time control systems,” *Advances in Computers* **49**, 303–352 (1999).
2. A. S. Tannenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms* (Prentice Hall, New York, 2002).
3. J. A. Stankovic and K. Ramamritham, “What is predictability for realtime systems?,” *Journal of Realtime Systems* **2**, 247–254 (1990).
4. J. A. Stankovic, “Real-time and embedded systems,” *ACM Computing Surveys* **28**(1), 205–208 (1996).
5. J. A. Stankovic, “Strategic directions in real-time and embedded systems,” *ACM Computing Surveys* **28**(4), 751–763 (1996).
6. H. Berger, *Automating with Step 7 in Stl, Simatic S7-300/400 Programmable Controllers: Simatic S7-300/400 Programmable Controllers* (John Wiley, New York, 2001).

7. H. Koeptz, "Event triggered versus time triggered realtime systems," *Lecture Notes in Computer Science, 563* (Springer-Verlag, Dordrecht, 1991).
8. J. Hill, "A software architecture supporting networked sensors," M.S. Thesis, University of California, Berkeley (2000).
9. M. D. Mesarovic and Y. Takahara, *Mathematical Theory of General Systems* (Academic Press, New York, 1974).
10. G. Le Lann, "Notes on future operating systems for realtime dependable distributed computing," *Lecture Notes in Computer Science, 563* (Springer-Verlag, Dordrecht, 1991).
11. M. Humphrey and J. A. Stankovic, "Predictable threads for dynamic, hard real-time environments," *IEEE Transactions on Parallel and Distributed Systems,* **10**(3), 281–296 (1999).
12. M. Joseph, *Real-time Systems Specification, Verification and Analysis*, http://www.tcs.com/techbytes/htdocs/book_mj.htm (2001).
13. P. Li and B. Ravindran, "Fast, best-effort realtime scheduling algorithms," *IEEE Transactions on Computers* **53**(9), 1159–1175(2004).
14. L. Sha, R. Rajkumar, J. Lohoczky, and K. Ramamritham, "Mode-change protocols for priority-driven preemptive scheduling," *Journal of Realtime Systems,* Vol **1**, 243–264 (1989).
15. S. S. Iyengar, S. Sastry, and N. Balakrishnan, "Foundations of data fusion for automation," *IEEE Control Systems Magazine* **6**(4), 35–41 (2003).
16. J. H. Christensen, "Basic concepts of iec 61499," Tech. Rep., www.holobloc.com (2003).
17. I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: A survey," *Computer Networks* **38**, 393–422 (2002).
18. D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing* **1**(1), 59–69 (2002).
19. J. Elson and D. Estrin, "Time synchronization for wireless sensor networks," *Proceedings of the 15th International Parallel and Distributed Processing Symposium.* (IEEE Computer Society, Washington, 2001).
20. J. Elson and K. Romer, "Wireless sensor networks: A new regime for time synchronization," Tech. Rep., University of California, Los Angeles (2002).
21. L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins, "A survey of configurable, component based operating systems for embedded applications," *IEEE Micro*, May–June, 54–68 (2001).
22. G. Denys, F. Pissens, and F. Matthijs, "A survey of composability in operating systems research," *ACM Computing Surveys* **34**(4), 450–468 (2002).
23. K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kahout, C. Smit, T. Zhang, and B. Jacob, "The performance and energy consumption of embedded real-time operating systems," *IEEE Transactions on Computers* **52**(11), 1454–1469 (2003).

Dr. Sastry is an Assistant Professor with the Department of Electrical and Computer Engineering, The University of Akron. He received his Ph.D. from Case Western Reserve University. In addition, he received a Master's degree in computer science from The University of Central Florida, a Master's degree in Electrical Engineering from the Indian Institute of Science and has worked in U.S. industry for about 12 years in areas of Automation, Knowledge Based Systems, and Software Engineering. His current research interests are in large-scale distributed systems, modeling, algorithms, and software engineering.



Dr. S. S. Iyengar is the Chairman and Roy Paul Daniels Chaired Professor of Computer Science at Louisiana State University and is also Satish Dhawan Chaired Professor at Indian Institute of Science. He has been involved with research in high-performance algorithms, data structures, sensor fusion, data mining, and intelligent systems. He has directed over 34 Ph.D. students, many of whom are faculty at major universities worldwide or scientists or engineers at national labs/industry around the world. His publications include 13 books (authored or coauthored, edited: Prentice-Hall, CRC Press, IEEE Computer Society Press, John Wiley & Sons, etc.) and over 300 research papers in refereed journals and conferences in areas of high-performance parallel and distributed algorithms and data structures for image processing and pattern recognition, and distributed data mining algorithms for biological databases. His books have been used by researchers at Purdue, University of Southern California, University of New Mexico, etc., at various times. His forthcoming book on distributed sensor networks will be released in October, 2004. He was a visiting professor at the Jet Propulsion Laboratory–Cal. Tech, Oak Ridge National Laboratory, the Indian Institute of Science, and at the University of Paris and other places. He has been on the prestigious National Institutes of Health-NLM Review Committee, in the area of Medical Informatics for 4 years. Dr. Iyengar is a Fellow of the Association of Computing Machinery (ACM), Fellow of the American Association of Advancement of Science (AAAS), Fellow of the Institute of Electrical and Electronics Engineering (IEEE), and has served on numerous panels for the U.S. National Science Foundation, National Research Council (Reviewed Proposals), the Defense Advanced Research Projects Agency, Member of the European Academy of Sciences, etc. He received the prestigious Distinguished Alumnus Award from Indian Institute of Science, Bangalore, in 2003. He has been the Program Chairman for many national and international conferences. He has given over 60 plenary talks and keynote lectures at numerous national and international conferences.