

State-Centric Programming for Sensor-Actuator Network Systems

Networked embedded systems must be able to rapidly respond to all kinds of sensing events, even though they have numerous failure-prone nodes and limited energy and bandwidth resources. The design methodology and software environment described here mediate between the developer's mental model of physical phenomena and the distributed application.

Distributed embedded systems such as wireless sensor and actuator networks require new programming models and software tools to support the rapid design and prototyping of sensing and control applications. Unlike centralized platforms and web-based distributed systems, these *distributed sensor-actuator network* (DSAN) systems are characterized by a massive number of potentially failing nodes, limited energy and bandwidth resources, and the need to rapidly respond to sensor input.

Jie Liu, Maurice Chu, Juan Liu,
James Reich, and Feng Zhao
Palo Alto Research Center

As application developers, we must fundamentally rethink the organization and programming of these deeply embedded systems. What is the appropriate mental model we can use to reason about the collective behaviors of a system when programming a distributed application so that the application is portable, scalable, and robust? What are the organizational principles for developers to build large applications by mixing and matching various ad hoc communication protocols while shielded from dealing with a multitude of communication events? How does the software architecture expose the underlying system constraints so that application developers can consider important performance trade-offs?

This article describes a state-centric, agent-based

design methodology to mediate between a system developer's mental model of physical phenomena and the distributed execution of DSAN applications. Building on the ideas of data-centric networking,¹ sensor databases,² and proximity-based group formation,³ we introduce the notion of *collaboration groups*, which abstracts common patterns in application-specific communication and resource allocation. An application developer specifies computations as the creation, aggregation, and transformation of states, which naturally map to the vocabulary used by signal processing and control engineers. More specifically, programmers write applications as algorithms for state update and retrieval, with input supplied by dynamically created collaboration groups. As a result, programs written in the state-centric framework are more invariant to system configuration changes, making the resulting software more modular and portable across multiple platforms. Using a distributed tracking application with sensor networks, we'll demonstrate how state-centric programming can raise the abstraction level for application developers.

Sensor network programming

Sensor-actuator network systems offer unique advantages over traditional centralized approaches. Dense networks of distributed sensors can improve perceived signal-to-noise ratio by reducing

average distances from sensor to physical phenomena. In-network processing and actuation shorten the feedback chain and improve the timeliness of observation and response. Untethered network nodes and infrastructureless mesh network topologies reduce deployment costs. However, the greatest advantages of networked systems are improved robustness and scalability. A decentralized system is inherently more robust against individual node or link failures because of network redundancy. Decentralized algorithms are also far more scalable in practical deployment; they might be the only way to achieve the large scales needed for some applications. Because of decentralized systems' spatial coverage and multiplicity in sensing aspect and modality, the detection, classification, and tracking of moving, nonlocal, or low-observable events require cross-node collaboration among sensors.

Collaborative signal and information processing (CSIP) algorithms, although having attractive properties such as robustness and scalability, are extremely hard to design, implement, and especially debug. The algorithms are highly specialized and must cope with a great deal of uncertainty—in physical hardware, sensor measurements, and communication. Traditional programming methodologies, such as node- and network-centric interfaces and imperative languages for networked embedded systems, don't provide adequate abstractions for CSIP applications. They provide system services such as networking, sensing, and task scheduling through a node-level operating system. Given the uncertainty and need to optimize across network and sensing layers, traditional methodologies push many quality-of-service decisions to the application developers. For example, because individual nodes and links are unreliable, providing reliable communications between sensors in the network stacks isn't always practical.

Most sensor network interfaces only provide primitives to discover communication peers and to send and receive messages in a best-effort manner. Reliable communications are implemented at the application level and must be backed up by fault-tolerant plans to cope with communication failures. As a result, applications are typically constructed as parallel finite-state machines running on each individual node to anticipate every

opment unlikely without significant improvements in our abstraction.

This complexity will be an even more serious obstacle for future sensor network developers, who likely will be domain experts rather than networking or operating system experts. (They think in terms of physical phenomena and signal processing instead of communication protocols and concurrency management.) Can we provide a programming model that

In most cases, only a relatively small subset of sensors contribute significantly to the estimation, owing to sensing-range limitations.

possible combination of concurrent sensing and communication events. The system's global behaviors are the result of these local FSMs' interactions. On the other hand, declarative interfaces such as SQL for databases provide a hardware-independent abstraction. However, CSIP application developers must still write collaborative processing programs to support the high-level declarative queries.

The framework we describe here fills this important space between high-level information processing and node-level executions. Node-centric programming abstractions for CSIP applications are fairly difficult to manage. In our own experiences with a collaborative single-target tracking system, nearly 40 percent of the code and most of the debugging effort dealt with communication and concurrency concerns, even though the system was built on well-designed communication protocols—directed diffusion and GEAR (*geographical and energy aware routing*).⁴ The FSM on each node quickly explodes the code size and complexity to respond to the various types and orders of occurrence of sensing and communication events. Implementing a multitarget tracker boosts complexity to a yet higher level, making practical devel-

opment unlikely without significant improvements in our abstraction. This complexity will be an even more serious obstacle for future sensor network developers, who likely will be domain experts rather than networking or operating system experts. (They think in terms of physical phenomena and signal processing instead of communication protocols and concurrency management.) Can we provide a programming model that

Target tracking as a motivating example

Tracking is a canonical problem for sensor networks and essential for many commercial and military applications such as traffic monitoring, facility security, and battlefield situational awareness. Given a moving point signal source or target in a 2D sensor field, a tracking system's goal is to estimate target state histories, such as spatial trajectory, on the basis of sensor measurements. From a tracking expert's point of view, each sensor node provides a local measurement useful in estimating the target state. However, in most cases, only a relatively small subset of sensors contribute significantly to the estimation, owing to sensing-range limitations. In this case, a good solution is a leader-based tracking scheme, such as *information-driven sensor querying*,^{5,6} to fuse information from only the sensors that provide high-quality measurements. As Figure 1 illustrates, at any time instant

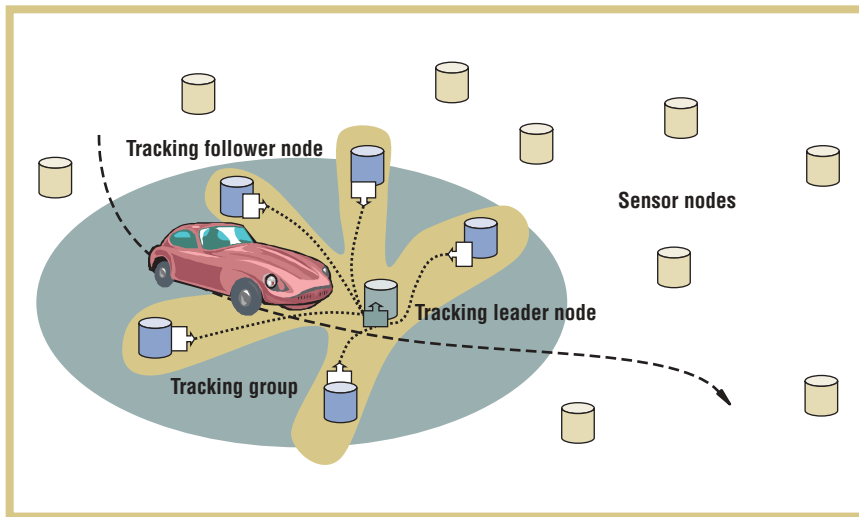


Figure 1. Collaborative processing in a leader-based object-tracking scenario. As a vehicle moves through a sensor field, nearby sensors detect it. An elected leader node aggregates data from the active sensors and migrates the information from node to node as the vehicle moves.

ers must explicitly write code to

- Maintain sensor connectivities in a neighborhood
- Discover the best node for handoff
- Invite neighbor nodes into the group
- Handle communication delays and failures

When application complexity increases, traditional programming abstractions don't scale up accordingly. For example, tracking multiple moving targets, as we'll describe later, presents additional challenges beyond the simple multiplicity of targets, such as the need to handle interactions between multiple tracking leaders as well as the underlying target-sensing behaviors in close proximity.

State-centric programming

CSIP applications, such as target tracking, are not generic distributed programs. Deeply rooted in these applications are the notion of states of physical phenomena and models of their evolution over space and time. We can represent some states centrally, as in the point target-tracking example, but must represent others in a distributed fashion, as in the contour-tracking case.

A distinct property of physical states, such as the location, shape, and motion of objects, is continuity in space and time. We typically handle the sensing and control of these states through sequential state updates. System theories, the basis for many signal processing and control algorithms, provide the following state-centric abstraction for state updating:

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

$$y_k = g(x_k, u_k) \tag{2}$$

t , IDSQ designates a single node, located close to the target, as *leader*. The leader node fuses these high signal-to-noise-ratio measurements and updates its current target location estimate, referred to as the *belief*. This can be done using, for example, sequential Bayesian filtering.

For most sensor types, owing to the physical properties of signal propagation, the sensors with high signal-to-noise ratio will be within a limited range of the leader node. So, you can minimize the communication cost and latency for gathering sensor data. As the target traverses the sensor field and the belief evolves to follow its motion, the most "informative" sensors might no longer be those closest to the current leader. A nearby sensor might then be selected to replace this leader on the basis of the updated belief and a criterion combining resource constraints with some measure of sensing utility (such as mutual information). The current leader then hands off the belief to this sensor, which becomes the next leader at time $t + \delta$, where δ is the communication delay. The process of sensing, estimation, and leader selection repeats.

We term the organization of sensing and the associated data transport tasks in CSIP applications the *models of collaboration* among the sensors. In addition to leader-based collaboration, there are other types of sensor collaboration. Consider another example where a sen-

sor network collaboratively finds a contour of constant temperature within the sensor field (see Figure 2). In this case, the state of the tracked phenomenon (that is, the shape of a specific temperature contour) might not be available to a single sensor or its local neighborhood. Each sensor node might detect a section of the contour by locally comparing sensor measurements, but the network must piece together these partial states to form the contour object's global state.

The types of sensor collaboration in these two examples are quite different. In the first example, the sensor nodes collaborate primarily to improve sensing accuracy, and acceptable estimation quality might be achieved using only a subset of the sensors. One node, the leader, plays a key role in fusing others' sensor measurements. In the contour-tracking example, on the other hand, no leader is present, and all sensors that form the contour are equally important. Each node might locally update and repair its observation of a contour section, but the global state can only be assembled from observations of many nodes along the entire contour.

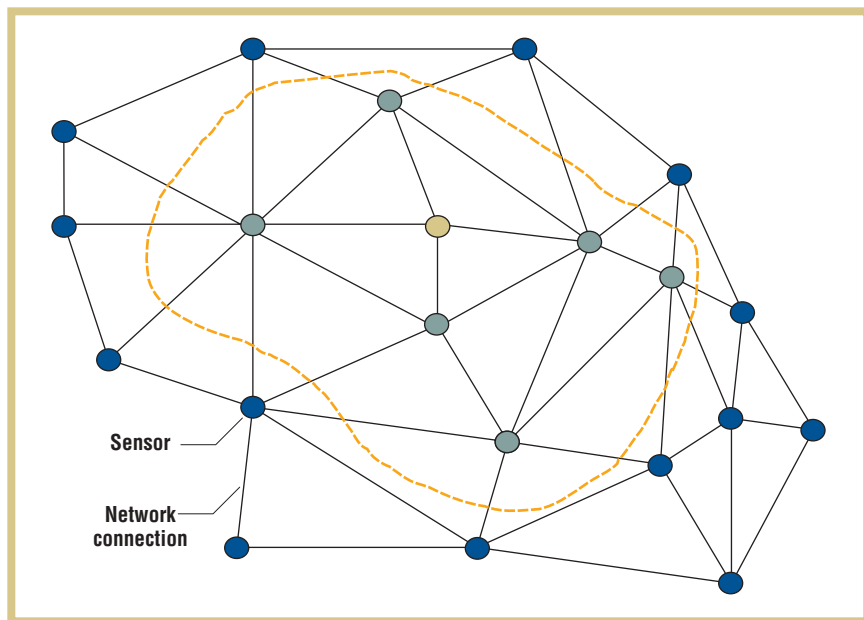
These high-level notions of collaboration, although useful as conceptual models for programmers to prototype CSIP applications, do not directly map to programs under traditional distributed-programming interfaces. System design-

Figure 2. Collaborative processing in detecting a contour of physical phenomena, such as a temperature field. The green nodes, which are inside and immediately adjacent to the contour, act as surrogates for the contour, each storing local information about contour crossings.

where x is the system state, k is an integer update index over space or time, u is input, y is output, f is the state update function, and g is the output or observation function. This formalization is broad enough to capture a wide variety of algorithms in sensor fusion, signal processing, and control (for example, Kalman filtering, Bayesian estimation, system identification, feedback control laws, automata, and so on). State-centric programming abstractions have been successfully applied to synchronous VLSI circuit designs and (centralized) control system designs. Synchronous languages such as Signal (www.irisa.fr/espresso/Polychrony) and Esterel (www.sop.inria.fr/esterel.org) and mixed-signal visual languages such as Matlab's Simulink (www.mathworks.com) and Ptolemy II's CT Domain (ptolemy.eecs.berkeley.edu) are all examples of state-centric programming models.

However, in a distributed real-time embedded system, the formulation is not as cleanly represented as in the abstraction just given. The relationship among subsystems can be highly dynamic. We must address concerns such as

- 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 it take to acquire the set of inputs?
- Are the inputs in u_k acquired synchronously?
- Do the inputs arrive in the correct order through communication?
- What is the choice of the update interval? Are they consistent?



System designers cannot be entirely shielded from these issues without seriously compromising system correctness and efficiency. These concerns address where and when, rather than how, to perform sensing, computation, and actuation, and play a central role in achieving the overall system performance. However, traditional programming models and languages don't support these "nonfunctional" aspects of computation (related to concurrency, reactivity, networking, and resource management) well. We need novel design methodologies and frameworks that provide meaningful abstractions for these issues, so that domain experts can continue to express algorithms and write programs in the style of these abstractions but still maintain an intuitive understanding of where and when to perform these operations. Domain-specific runtime systems are to support this design methodology to ensure correct and efficient execution and allow transparent layering-in of features such as security and reliable communication.

Collaboration groups

The abstraction we provide is the notion of *collaboration groups* (or *groups* for short). A group is a set of entities that

contribute to a state update. These entities can be physical or logical sensor nodes or more abstract system components such as software agents. In this context, we refer to them all as *agents*.

Intuitively, a group encapsulates two properties: its *scope* and its *structure*. A group's scope defines its members. You can specify a scope existentially or by a membership function (for example, all nodes within some geometric range, all nodes within a certain number of hops from an anchor node, or all nodes that are "close enough" to a temperature contour). Grouping nodes according to physical attributes rather than node addresses is an important abstraction in sensor network programming. You can evaluate scope locally or dynamically, as long as you maintain communication among the group members as needed. Scoping is critical to maintain scalability as sensor networks grow arbitrarily large.

A group's structure defines the roles each member plays in the group. Do all group members have equivalent roles? Is there a "leader" member that consumes data? Do group members form a tree with parent and children relations? The notion of roles shields programmers from addressing individual nodes by either name or address. In fact, in most

sensor network applications, neither name nor address is important. Redundancy by having multiple members with the same role also improves application robustness over node and link failures. Another advantage of the notion of roles is that you can induce communication and control patterns from them, so that you can implement highly optimized networking and control flow protocols at runtime to support group execution.

Formally, a group is a 4-tuple

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

where A is a set of agents; L is a set of labels, called roles; $p : A \rightarrow L$ is a function

Having multiple members with the same role also improves application robustness over node and link failures.

that assigns each agent a role label; and $R \subseteq L \times L$ are the connectivity relations between roles. Given the relations between roles, a group can induce a lower-level connectivity relation E among the agents such that $\forall a, b \in A$, if $(p(a), p(b)) \in R$, then $(a, b) \in E$. For example, under this formulation, the leader-follower structure defines two roles $L = \{leader, follower\}$ and a connectivity relation $R = \{(follower, leader)\}$, meaning that the follower sends data to the leader. Then, by specifying one leader agent and multiple follower agents within a geographical region (that is, specifying a map p from a set of agents A to labels in L), we have effectively specified that all followers send data to the leader without addressing the followers individually.

At runtime, the scope and structural dynamics of groups are managed by group management protocols, which are highly dependent on the types of groups. A detailed specification of group management protocols is beyond this article's

scope, but we can provide examples and discuss several protocols at a high level.

Examples of groups

Combinations of scopes and structures create generic patterns of groups that could be highly reusable from application to application. Here, we give a few examples of groups to illustrate their wide variety and the importance of being able to mix and match them in applications. We classify the groups by their scopes. Within each type of group, there might be more than one structure. For example, a group defined by a geographical region might have a single leader (that is, a leader-follower group), multi-

ple concurrent leaders (a multileader group), or no leader at all.

Geographically constrained group (GCG). A GCG consists of a set of nodes located within some fixed geographical region (the group's *extent*). Because physical signals, especially those from point targets, usually attenuate with distance, this type of group naturally captures the set of sensors that can possibly sense a local phenomenon. There are many ways to specify the GCG's geographic shape, such as circles, polygons, and their unions and intersections. You can easily establish a GCG by geographically constrained flooding, using protocols such as Geocasting⁷ and GEAR to support communication among members. A GCG may designate a leader, which fuses information from all other group members.

N-hop neighborhood group (n-HNG). When the communication topology is more important than the geographical

extent, we can define scope in terms of hop counts. We define the members of an n -HNG as the set of all nodes within n communication hops from a specified anchor node. Because local broadcasting uses hop counts rather than Euclidean distances, it can be conveniently used to determine the scope. Usually, the anchor node is the group leader. The group might have a tree structure with the leader as the root to optimize communication. If you can decompose the leader's behavior into suboperations running on each node, then the tree structure also provides a platform for distributing the computation.

There are several useful special cases for n -HNG. For example, 0-HNG contains only the anchor node itself, 1-HNGs are the anchor node's one-hop neighbors, and ∞ -HNG contains all the nodes reachable from the root. From this point of view, TinyDB is built on an ∞ -HNG group.²

Publish/subscribe group (PSG). You can also define a group more dynamically, such as all entities that can provide certain data or services or that satisfy certain predicates over its observations or internal states. A PSG consists of one or more consumers expressing interest in specific types of data or services and the zero or more producers that provide those. You can establish a PSG via rendezvous points, directory servers, or network protocols such as directed diffusion.⁸

Acquaintance group (AG). Even more dynamic is the AG, where a member belongs to the group because another group member "invited" it. The relationships among the members might be purely logical and historical rather than depending on any physical properties at the time. A member may quit the group without needing any other member's permission but might have to "clean up

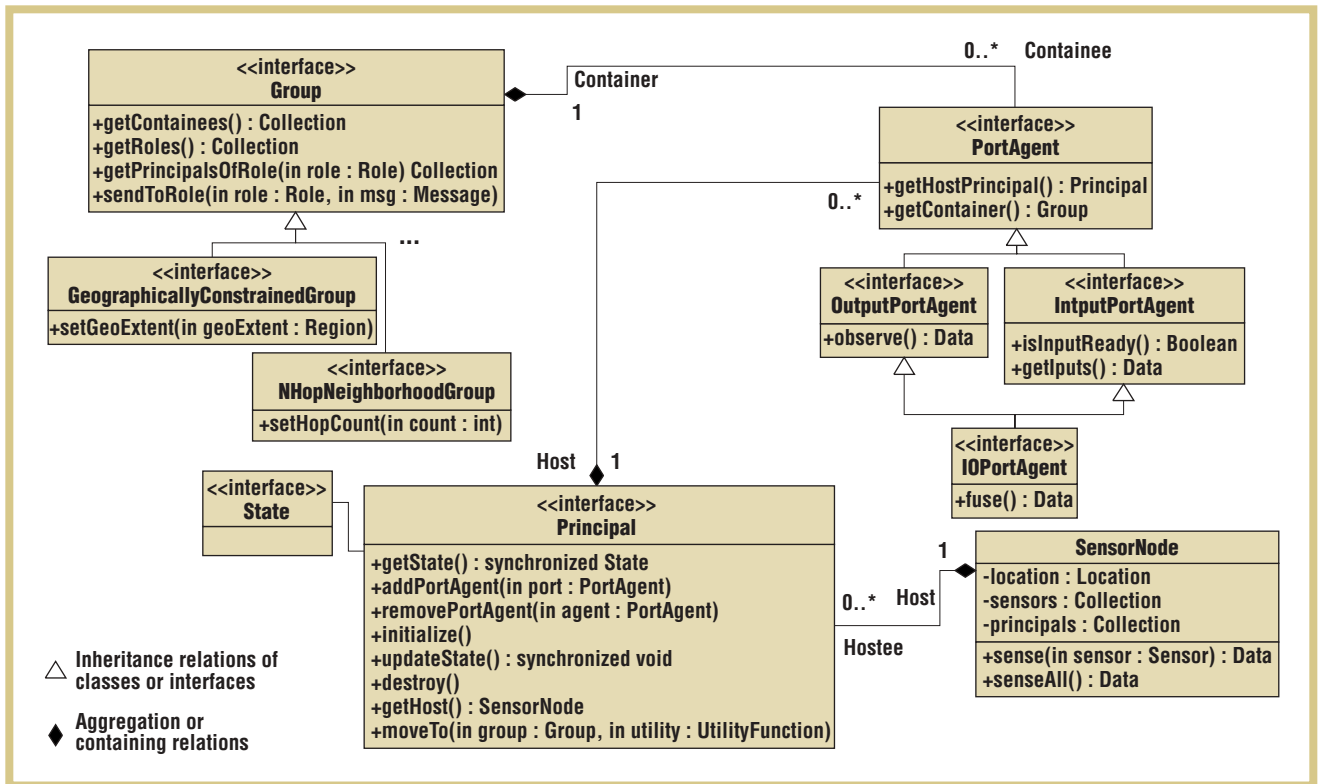


Figure 3. A UML diagram for the base classes and interfaces in PIECES. A principal may host zero or more port agents, while each port agent is contained by exactly one group.

after itself” to keep from severing other members from the group. An AG might have a leader serving as the rendezvous point. When the leader is also fixed on a node or in a region, GPSR (*greedy perimeter stateless routing*),⁹ ad hoc routing trees, or directed-diffusion types of protocols can facilitate communication between the leader and the other members. An obvious use of this group is to monitor and control mobile agents from a base station. When all group members are mobile and no central leader exists, maintaining group connectivity might become quite complex.

Using multiple groups

Despite the fact that there are efforts to provide a single unified collaboration model for all DSAN applications, we believe that mixing and matching multiple group classes is a more powerful technique to tackle system complexity, matching information flow closely to the

algorithm designer’s conceptual model of information flow without sacrificing scalability or efficiency. Even within a particular type of group, you can select highly tuned communication protocols for a specific application to reduce latency and energy cost while maintaining the same clear mental model of information flow.

There are various ways to compose groups. You can do so in parallel to provide different types of input for a single computational entity. For example, in the previous point target-tracking example, you could use a GCG to gather sensor measurements and a 1-HNG to select the potential next leader. You can also compose groups hierarchically such that one group contains another (or its representative member). For example, while using multiple groups to compute target trajectory, all tracking leaders of various targets might form a PSG with the base station to report tracking results.

A state-centric design framework

We implement the methodology of state-centric programming over collaboration groups in a PARC-developed software environment called PIECES (Programming and Interaction Environment for Collaborative Embedded Systems) to help model, simulate, design, and deploy sensor network applications. So far, we’ve supported the modeling and simulation capabilities of PIECES in a mixed Java-Matlab environment, and we’ve demonstrated these capabilities on a challenging multitarget-tracking simulation with realistic sensor and environment models. We’re still designing a runtime execution platform that directly supports our methodology.

Programming model

Figure 3 shows the basic entities in PIECES and their relationships. In PIECES, programmers think in terms of dividing

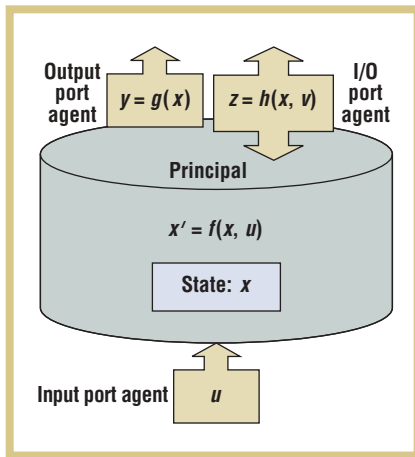


Figure 4. A principal and its port agents in PIECES. The principal maintains state x and updates it through the evaluation of f . The input port agent provides the input data for this update. Output port agents or I/O port agents can be attached to the principal to produce $g(x)$ or $h(x, v)$.

the global state of physical phenomena into a hierarchical set of independently updatable pieces with one computational entity (called a *principal*) maintaining each piece. To update the state, principals can request inputs from other principals, with sensing principals supporting the lowest-level sensing and estimation tasks. Communication patterns are specified by defining collaboration groups over principals and assigning corresponding roles for each principal through ports. Principals may move from node to node to improve performance and reduce the cost of sensing, computation, and communication. With group-level support, application developers can focus on implementing the state update functions as if they were centralized programs.

Principals and port agents. The framework comprises two key components, principals and port agents, as Figure 4 shows.

A principal encapsulates a piece of *state*. Typically, these states correspond to certain aspects of the physical phenomena of interest. (From a computational perspective, a port agent as an object certainly has its own state, but the distinction here is that these states are not associated directly with physical phenomena.) The role of a principal is to update its state from time to time, a computation corresponding to evaluating function f in Equation 1. A principal also accepts other principals' queries of certain views on its own

state, a computation corresponding to evaluating function g in Equation 2.

To update its portion of the state, a principal can gather information from other principals. When a principal needs information from other principals, it creates port agents and attaches them onto itself and onto the other principals. A port agent can be an input, an output, or both. An output port agent is also called an *observer*, because it computes outputs on the basis of the host principal's state and sends them to other agents. Observers might be active, pushing data autonomously to its destinations, or passive, sending data only when the consumer sends a query. A principal typically attaches a set of observers to other principals, and creates a local input port agent to aggregate the information collected by the remote agents. Thus, port agents capture communication patterns among principals.

Note the separate executions of principals and observers. Principals maintain states, reflecting 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 must 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 isn't currently needed, there's 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 S , can only be maintained by exactly one principal. This doesn't prevent other principals from having local caches of S for efficiency and performance reasons,

nor does it prevent the other principals from locally updating the values of cached S . However, there is only one "master copy" for S , all local updates should be treated as "suggestions" to the master copy, and only the principal that owns S has the final word on its values. This asymmetric access of variables simplifies the way shared variables are managed.

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

An application can delegate group creation to port agents, especially for leader-based groups. The leader port agent, typically of type input, can be created on a principal, and the port agent can take group scope and structure parameters to find the other principals and create follower port agents on them. You can create groups dynamically on the basis of principals' collaboration needs. For example, when a principal responsible for tracking a target finds more than one target in its sensing region, it can create a classification group to fulfill the need for classifying targets. A group might have a limited time span. When certain collaborations aren't needed, their corresponding groups can be deleted.

A group's structure allows its members to address other principals through their roles rather than their names or logical addresses. For example, the only interface that a follower port agent in a leader-follower structured group needs is to send data to the leader. If the leader moves to another node while a data packet leaves a follower agent, the group management protocol should take care of the dangling packet, either delivering it to the leader at the new location or simply discarding it. This kind of leader-follower

group management protocol is built on top of data-centric routing and storage services such as diffusion routing and geographic hashing tables,¹⁰ whose implementation details are beyond the scope of this article.

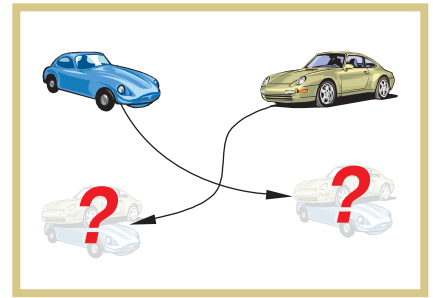
Mobility. A principal is hosted by a specific network node at any given time. The most primitive type of principal is a *sensing principal*, which is fixed to a sensor node. A sensing principal maintains a piece of (local) state related to the physical phenomena, based solely on its own local sensor measurement history. Although a sensing principal is constrained to never move, other principals might be implemented as software agents that move from host to host, depending on information utility, performance requirements, time constraints, and resource availability. A principal P might also be *attached* to another principal Q in the sense that P moves with Q . When a principal moves, it carries its state to the new location, and the scopes of the groups it belongs to change accordingly.

Mobile principals bring additional challenges for maintaining the state. For example, a principal should not move while it is in the middle of updating the state. To ensure this, PIECES restricts that whenever an agent is triggered, its execution must reach a quiescent state without further blocking on either input data or computation or sensing resources. Such a trigger is called a *responsible trigger*.¹¹ Only at these quiescent states can principals move to other nodes in a well-defined way, carrying the minimum amount of information representing the phenomena.

Simulator

PIECES provides a mixed-signal simulator that simulates sensor network applications at a high level. We implemented the simulator using Java and Matlab and built an event-driven engine in Java to simulate network message passing and

Figure 5. Data association problem: as multiple objects come close to each other, the sensor system must properly associate measurements with hypotheses of objects to identify them. Without discriminatory characteristics such as color, resolving whether the vehicles have crossed paths or merely passed by each other is impossible.



agent execution at the collaboration group level. Our continuous-time engine, built in Matlab, simulates target trajectories, signals, noise, and sensor front ends. The main control flow is in Java, which maintains the global notion of time. The interface between Java and Matlab also makes it possible to implement functional algorithms such as signal processing and sensor fusion in Matlab, while leaving their execution control in Java. A three-tier distributed architecture is designed through Java registrar and remote-method-invocation interfaces, so that the execution in Java and Matlab can be separately interrupted and debugged.

Like most network simulators such as ns-2 (www.isi.edu/nsnam/ns), the PIECES simulator maintains a global event queue and triggers computational entities—principals, port agents, and groups—via timed events. However, unlike network simulators that aim to accurately simulate network behavior to the packet level, the PIECES simulator verifies CSIP algorithms in a networked execution environment at the collaboration group level. Although groups must have distributed implementations in real deployments, they are centralized objects in the simulator. They can internally use instant access to any member of any role, although these services are not available to either principals or port agents. This relieves the burden of developing, optimizing, and testing all the communication protocols jointly with the CSIP algorithms.

The PIECES simulator estimates communication delays by instantaneously checking the sender's and receiver's locations and computing a randomized delay corresponding to the group management

protocol. For example, if an output port of a sensing principal calls `sendToLeader(message)` on its container group, then the group obtains the sensor nodes that host the sensing principal and the destination principal, computes the number of hops between the two nodes specified by the group management protocol, and generates a randomized delay and a bit error based on the number of hops.

Example: A multitarget tracking system

We implemented a multiple-target-tracking algorithm with classification and identity management to demonstrate how state-centric programming supports and modularizes the software design of DSAN applications. You can find details of the algorithm with classification¹² and of identity management¹³ elsewhere.

Tracking, classification, and identity management

Multitarget tracking is the problem of estimating the trajectories of multiple targets from a time series of sensor measurements, possibly from multiple sensors. In single-target tracking, the problem is usually posed as a dynamic estimation problem of a partially observable Markov process, and a Bayesian recursive filter provides the solution. In multiple-target tracking, in addition to the estimation problem of tracking each individual target, there is the additional difficulty of identifying which track estimate should be associated with which target.

To illustrate this *data association* problem, we consider the scenario shown in Figure 5. When two targets come near

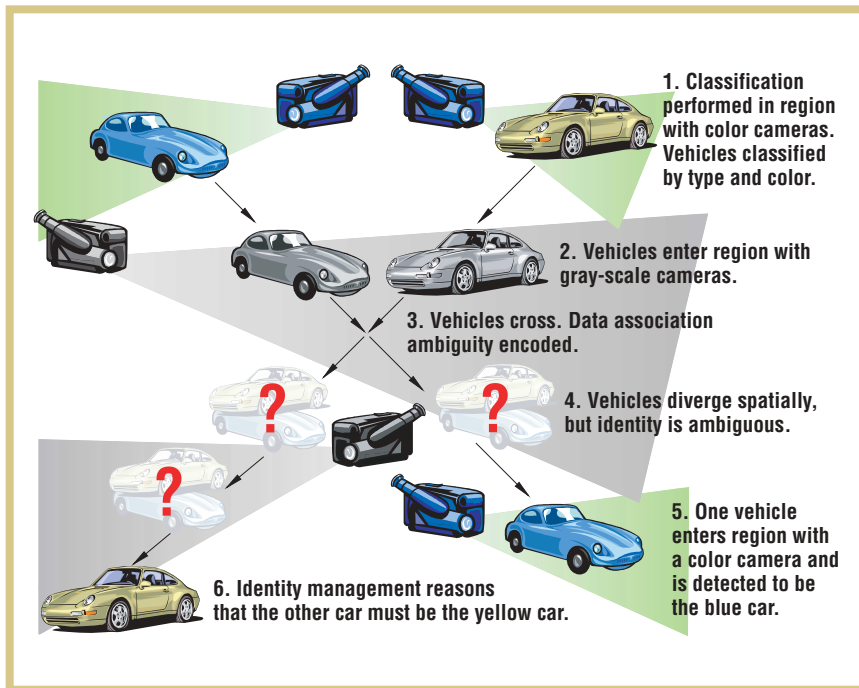


Figure 6. Identity management scenario: when two vehicle trajectories cross under a gray-scale camera, their identities are mixed. When we expose one vehicle to a color camera to reveal its identity, we also determine the other vehicle's identity.

sors near the targets. We can update the identity information by either local classification or identity management across tracks. We have designed the PIECES system as a set of communicating multiple target trackers (MTTrackers), where each tracker maintains the trajectory and identity information about a target or a set of spatially adjacent targets. An MTTracker is implemented by three principals: a *tracking principal*, a *classification principal*, and an *identity management principal* (see Figure 7).

A tracking principal updates the track position state periodically. It uses a GCG with a leader-follower relation to collect primitive local-position estimates from other sensors close to the target. The tracking principal is the leader, and all sensing principals within a certain geographical extent centered around the current target position estimate are the followers. The tracking principal also makes hopping decisions on the basis of its current position estimate and the node characteristic information collected from its one-hop neighbors via a 1-HNG.

Figure 8 shows sections of the *Tracking-Principal* code in PIECES. When the principal is initialized, it creates the agents and corresponding groups. Behind the scene, the groups create follower agents with specific types of outputs, indicated by the sensor modalities. Without further instructions from the programmer, the followers periodically report their outputs to the input port agents. Whenever the principal is activated by a time trigger, it updates the target position using the newly received data from the followers and selects the next hosting node on the basis of neighbor node characteristics.

Both the classification principal and the identity management principal operate on the identity state, with the identity man-

each other in space, an ambiguity might occur: either the targets actually cross paths or they approach each other and then diverge. Without further evidence regarding the target identities, the best we can do in this case is to explicitly quantify the degree of ambiguity so that the tracker doesn't falsely conclude that the two targets have either crossed paths or merely passed by each other.

In practical situations, we can often distinguish two targets from each other on the basis of their signal features. A *classification* algorithm can extract discriminatory information and make inferences about target identity. With a classifier's help, an *identity management* algorithm can help resolve data association ambiguities. Figure 6 presents an example scenario. When targets diverge, the tracking system can collect classification evidence. Such evidence should update the identity of both the local track and the remote track with which the local track has been mixed. Identity management is responsible for maintaining this global consistency of identities among tracks.

Implementing such a system using a node-centric programming interface is

extremely complicated. We can design the trackers as mobile agents that follow targets.⁴ However, each tracker must respond to many kinds of events, such as aggregated sensor data from nearby nodes, handoff messages from one or more track leaders, suppression messages to prevent it from initiating duplicate targets, classification data from different kinds of sensors, identity information requests from other agents, identity update information from other agents, multiple kinds of handshaking messages, and so on. Worse, these messages are interleaved because of agents' concurrent progression and the communication uncertainties. The logic that reacts to these messages must cover all the combinations of ordering of events. If the agents are built using flat FSMs, the number of states and transitions quickly becomes unmanageable.

Distributed implementation in PIECES

In terms of the states of the physical phenomena, the user cares about two kinds of target information in this application: the targets' positions and identities. We can estimate positions by fusing the energy measurements from the sen-

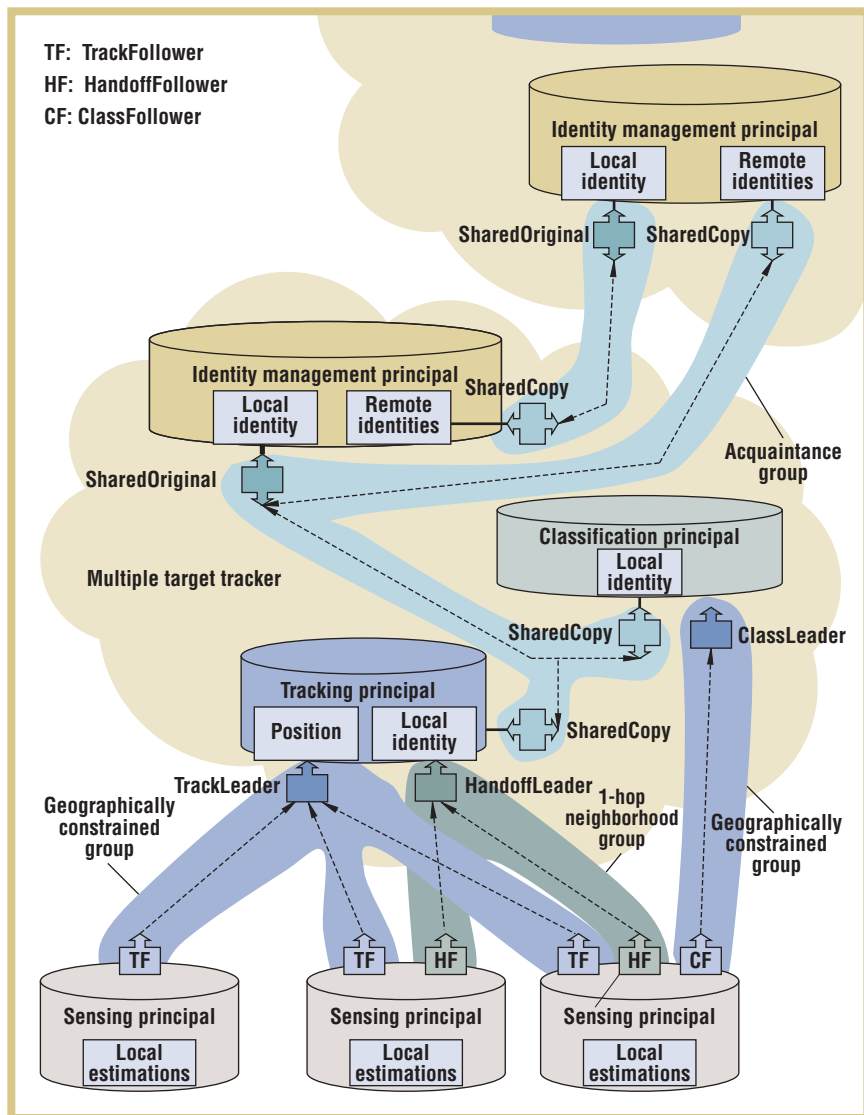
Figure 7. The distributed multiobject tracking algorithm as implemented in the state-centric programming model, using distributed principals and agents. Notice that the state-centric model lets developers focus on key pieces of state information that the sensor networks are trying to create and maintain, thus raising programming's abstraction level.

agement principal maintaining the “master copy.” In fact, an MTTracker creates the classification principal only when targets must be classified, and it “suggests” its computational results to the identity management principal. The classification principal uses a GCG to collect class feature information from nearby sensing principals in the same way that tracking principals collect location estimates.

The identity management principal forms an AG with all other identity management principals that might have relevant identity information. They become the member of a particular identity group only when targets intersect and their identities mix. Both classification principals and identity management principals are attached to the tracking principal for their mobility decisions. However, the formation of an AG among these three principals also provides the possibility that they can make their own hopping decisions without changing their interaction interface.

Simulation

Figure 9 shows the progression of tracking two crossing targets. The small squares represent sensor nodes, the lines represent vehicle trajectories, colored dots represent the current beliefs of vehicle positions, and the vehicles' identities are depicted using green and red. Initially, when the targets are well separated, as in Figure 9a, each target is tracked by a tracker whose sensing group is pictured as a shaded disk. The hosting node of the tracking principal is plotted in light blue, and the hosts for corresponding sensing principals are plotted in white. Since the targets are



well separated, each identity group contains only one member, the identity management principal of a tracker. As the targets move toward the center of the sensor field, the sensing groups move with their respective track positions.

In Figure 9b, the two separate tracking groups have merged. A joint tracking principal, now hosted by the yellow node, updates tracks for both targets. The system merges the tracks because tracking the targets jointly, rather than independently, when they approach each other is more accurate owing to the effect of signal superimposition. Finally, as the targets diverge, the merged tracking group and tracking principals split into two,

and each tracking principal proceeds to track one target separately as shown in Figure 9c. At this point, the targets' identities are mixed, so the system creates an identity group containing the two trackers' identity management principals.

Figure 10 shows a snapshot of a more complicated multitarget crossover scenario. Three tracks (whose identities are depicted using red, green, and yellow respectively in Figure 10a) have crossed each other in the past, so their identities are mixed. The corresponding identity management principals form an identity group. At the time of this snapshot, the identity management principal in the bottom-right corner is collecting classi-

```

public class TrackingPrincipal extends MobilePrincipal {
    ...
    private InputPortAgent _sensingAgent;
    private InputPortAgent _handoffAgent;
    private BeliefState _trackBelief;
    private UtilityFunction _utilityFunction = new InformationUtility();
    ...
    public void initialize() {
        // This principal is the client of the sensing agent.
        _sensingAgent = new InputPortAgent(this);
        // Create a GCG with a geometric extent to specify the scope
        // the _sensingAgent as the leader, a follower sensing type,
        // and a follower report period.
        GeoConstrainedLeaderGroup sensingGroup = new
            GeoConstrainedLeaderGroup(geometricExtent, _sensingAgent,
            energySensorClass, reportPeriod1);

        //This principal is the client of the handoff agent.
        _handoffAgent = new InputPortAgent(this);
        // Create a 1-HNG with the _handoffAgent as the leader,
        // a follower sensing type, and a follower report period.
        HopNeighborhoodGroup handoffGroup = new
            HopNeighborhoodGroup(1, _handoffAgent,
            nodeCharacteristicsSensorClass, reportPeriod2);
    }

    // The principal is awakened every 1 second by a time trigger.
    public void react(WakeupEvent event) {
        // Compute the new target belief state.
        updateState();
        // Choose the next host of this principal.
        moveTo(_handOffGroup, _utilityFunction);
    }

    // Key function to update the state of the track.
    public synchronized void updateState() {
        if(_sensingAgent.isInputReady()) {
            _trackBelief = track(_trackBelief, _sensingAgent.getInputData());
        }
    }
    ...
}
    
```

Figure 8. Sections of `TrackingPrincipal` code illustrating the creation of multiple groups to collect inputs for the tracking principal. PIECES computes the state estimate and then inserts it into the `track()` function without considering where the inputs are collected from.

What should be the appropriate programming models for DSAN applications? Is a network simply wireless “wires” that can ship one sensor’s reading to another? Is a sensor network a distributed database, reducing programs to filters and queries on the data? We have presented a methodology that views a sensor network as a distributed platform for in-network signal and information processing. Mobile principals, each maintaining a portion of the state of the physical phenomena of interest, interact with each other through collaboration groups. Having well-encapsulated state variables updated by the execution of principals gives designers a natural abstraction that is close to those of system theories. Mobility lets principals follow the phenomena spatially, minimizing latency and resource consumption. The abstraction of collaboration groups shields designers from complicated communication protocol management and event-handling issues while supporting a rich variety of models of cross-node interaction.

We’re still implementing the state-centric programming methodology in PIECES, especially the specific types of group interaction protocols, the resource allocation mechanisms, and the runtime support systems. However, we believe that this is an important first step in formulating and understanding the models of collaboration among distributed agents and the programming of collectives for sensing, computation, and control applications in DSAN systems. ■

fication information and identifying the track as the red target. So, it communicates with its peers in the identity management group (top-right and bottom-middle principals) to update their respective targets’ identities as well. The bar chart in Figure 10b shows the updated identity. Defining the acquain-

tance group and its interface in this way allows these spatially distributed identity management principals to communicate with each other. This gives developers the necessary abstraction to focus on the functional aspect of identity management algorithms without worrying about communication details.

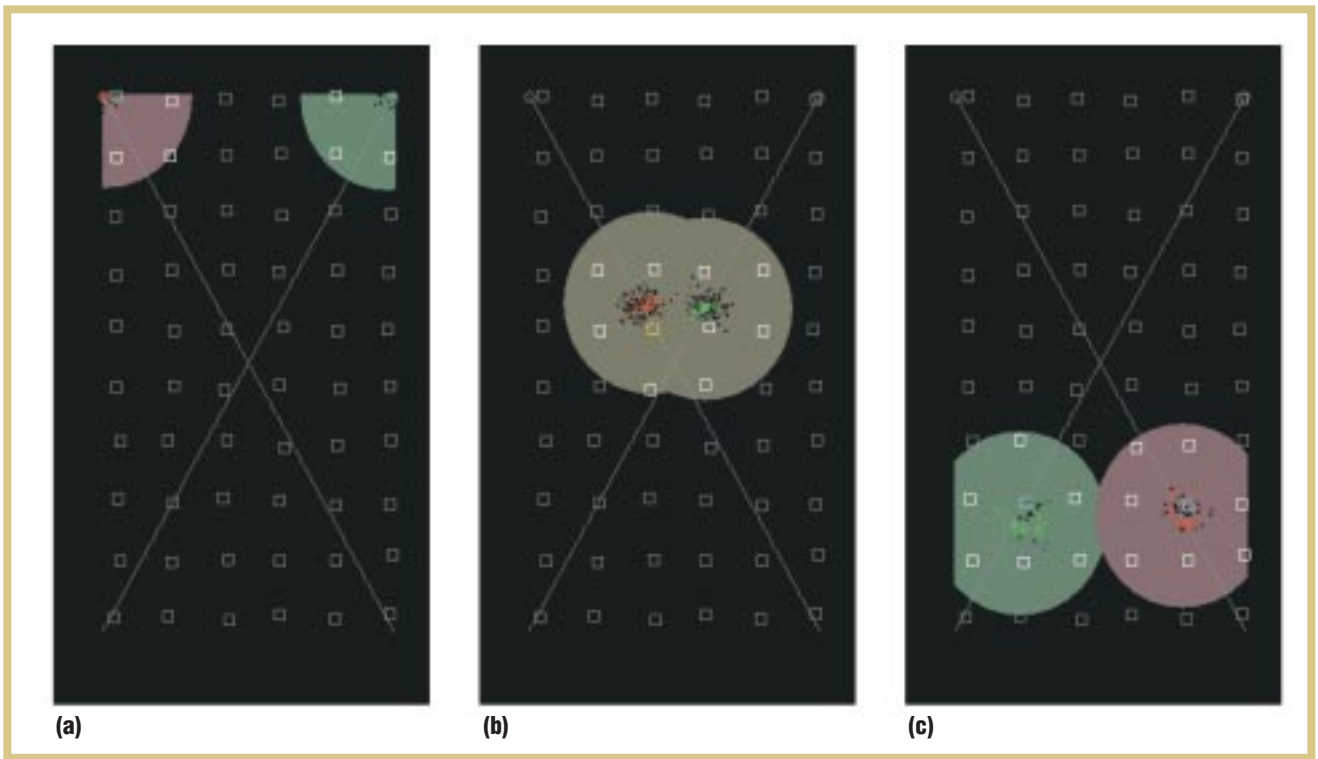
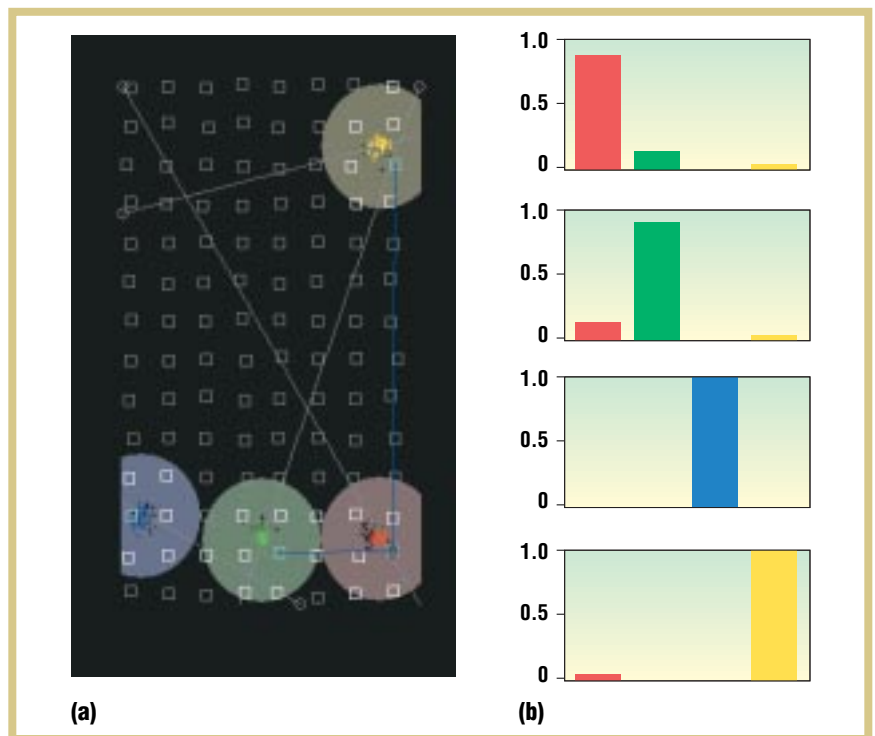


Figure 9. Tracking two simulated crossing targets. The small squares represent sensor nodes, the lines represent vehicle trajectories, colored dots represent the current beliefs of vehicle positions, and the vehicles' identities are depicted using green and red. (a) Pieces creates the geographical constrained group for each target. (b) When the two targets cross, their groups merge. (c) The group splits as the targets diverge.

ACKNOWLEDGMENTS

This work is partly supported by DARPA under contract F30602-00-C-0139 through the Sensor Information Technology Program. We also thank Dan Larner and Jaewon Shin for inspiring discussions and help in implementing the multitarget-tracking example discussed here.

Figure 10. Simulation results for tracking four moving targets. After the red, green, and yellow targets mixed in the past, the acquaintance group is established for identity management. When new identity information is collected, a distributed renormalization across this group resolves the target identities. (a) The tracker group and tracking results. (b) Bar charts of each track's identity after renormalization.



the AUTHORS



Jie Liu is a member of the research staff at the Palo Alto Research Center. His research interests include software architectures; programming models; simulation and synthesis tools; ad hoc networking; mixed-signal and hybrid systems; and their application in modeling, simulation, and design of distributed real-time embedded systems. He received his PhD in electrical engineering and computer sciences from the University of California, Berkeley. He is a member of the IEEE and the ACM. Contact him at PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; jieliu@parc.com.



Maurice Chu is a member of the research staff at the Palo Alto Research Center in the Embedded Collaborative Computing area of the Systems and Practices Laboratory, where he investigates the development of efficient inference and estimation applications for sensor network technology. He received his PhD in electrical engineering and computer science from MIT. His research interests include distributed information processing, communications and information theory, graphical models, learning and inference, and recognition in vision systems. Contact him at PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; mchu@parc.com.



Juan Liu is a member of the research staff at the Palo Alto Research Center. Her research interests include signal processing, statistical modeling, detection and estimation, network routing, information theory, and their applications to distributed sensor networks. She received the 2002 IEEE Signal Processing Society Best Paper Award for Young Authors and is a member of the IEEE. She received her PhD in electrical engineering from the University of Illinois, Urbana-Champaign. Contact her at PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; juan.liu@parc.com.



James Reich is a member of the research staff at the Palo Alto Research Center. His research focuses on scalable collaborative signal processing approaches and programming models for real-time in-network processing of sensor data. He received his MS in electrical and computer engineering from Carnegie Mellon University. He is a member of the IEEE. Contact him at PARC, 3333 Coyote Hill Rd., Palo Alto, CA 94304; reich@parc.com.



Feng Zhao is a principal scientist at the Palo Alto Research Center, where he directs the Embedded Collaborative Computing Area in the Systems and Practices Laboratory. He is also a consulting associate professor of computer science at Stanford University. His research interests include networked embedded systems, model-based diagnosis, qualitative reasoning, control of dynamical systems, and programming tools. He is the founding editor in chief of the *ACM Transactions on Sensor Networks*. He received his PhD in electrical engineering and computer science from MIT. Contact him at PARC, 3333 Coyote Hill Road, Palo Alto, CA 94304; fz@alum.mit.edu.

REFERENCES

1. D. Estrin et al., "Next Century Challenges: Scalable Coordination in Sensor Networks," *Proc. 5th Ann. ACM/IEEE Int'l Conf. Mobile Computing and Networking (ACM/IEEE MobiCom 99)*, ACM Press, 1999, pp. 263–270.
2. S. Madden, M.J. Franklin, and J.M. Hellerstein, "Tag: A Tiny Aggregation Service for Ad-Hoc Sensor Networks," *Proc. 5th Symp. Operating Systems Design and Implementation (OSDI)*, USENIX, 2002.

3. J. Liu et al., "Distributed Group Management for Track Initiation and Maintenance in Target Localization Applications," *Proc. 2nd Int'l Workshop Information Processing in Sensor Networks (IPSN 03)*, LNCS 2634, Springer-Verlag, 2003.
4. Y. Yu, R. Govindan, and D. Estrin, *Geographical and Energy Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks*, tech. report UCLA/CSD-TR-01-0023, Computer Science Dept., Univ. of California, Los Angeles, 2001.

5. M. Chu, H. Haussecker, and F. Zhao, "Scalable Information-Driven Sensor Querying and Routing for Ad Hoc Heterogeneous Sensor Networks," *Int'l J. High-Performance Computing Applications*, vol. 16, no. 3, Fall 2002, pp. 90–110.
6. J. Liu, J.E. Reich, and F. Zhao, "Collaborative In-Network Processing for Target Tracking," *EURASIP J. Applied Signal Processing*, vol. 2003, no. 4, Mar. 2003, pp. 378–391.
7. Y.-B. Ko and N.H. Vaidya, "Geocasting in Mobile Ad Hoc Networks: Location-Based Multicast Algorithms," *Proc. IEEE Workshop Mobile Computer Systems and Applications*, IEEE Press, 1999.
8. C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks," *Proc. 6th Ann. Int'l Conf. Mobile Computing and Networking (MobiCom 00)*, IEEE Press, 2000.
9. B. Karp and H.T. Kung, "Greedy Perimeter Stateless Routing for Wireless Networks," *Proc. 6th Ann. Int'l Conf. Mobile Computing and Networking (MobiCom 00)*, IEEE Press, 2000.
10. S. Ratnasamy et al., "GHT: A Geographic Hash Table for Data-Centric Storage," *Proc. 1st ACM Int'l Workshop Wireless Sensor Networks and Applications (WSNA 2002)*, Sept. 2002, pp. 78–87.
11. J. Liu, *Responsible Frameworks for Heterogeneous Modeling and Design of Embedded Systems*, PhD dissertation, Univ. of California, Berkeley, 2001.
12. M. Chu, S. Mitter, and F. Zhao, "Distributed Multiple Target Tracking and Data Association in Ad Hoc Sensor Networks," *Proc. 6th Int'l Conf. Information Fusion*, 2003.
13. J. Shin, L. Guibas, and F. Zhao, "A Distributed Algorithm for Managing Multi-Target Identities in Wireless Ad-Hoc Sensor Networks," *Proc. 2nd Int'l Workshop Information Processing in Sensor Networks*, 2003.

For more information on this or any other computing topic, please visit our Digital Library at <http://computer.org/publications/dlib>.