PIM: A Novel Architecture for Coordinating Behavior of Distributed Systems

Kenneth M. Ford, James Allen, Niranjan Suri, Partrick J. Hayes, and Robert A. Morris

■ *Current approaches to the problem of coordinating the activity of physically distributed systems or devices all have well-recognized strengths and weaknesses. We propose adding to the mix a novel architecture, the process-integrated mechanism (PIM), that enjoys the advantages of having a single controlling authority while avoiding the structural difficulties that have traditionally led to the rejection of centralized approaches in many complex settings. In many situations, PIMs improve on previous models with regard to coordination, security, ease of software development, robustness, and communication overhead. In the PIM architecture, the components are conceived as parts of a single mechanism, even when they are physically separated and operate asynchronously. The PIM model offers promise as an effective infrastructure for handling tasks that require a high degree of time-sensitive coordination between the components, as well as a clean mechanism for coordinating the high-level goals of loosely coupled systems. The PIM model enables coordination without the fragility and high communication overhead of centralized control, but also without the uncertainty associated with the system-level behavior of a multiagent system (MAS). The PIM model provides an ease of programming with advantages over both multiagent systems and centralized architectures. It has the robustness of a multiagent system without the significant complexity and overhead required for interagent communication and negotiation. In contrast to centralized approaches, it does not require managing the large amounts of data that the coordinating process needs to compute a global view. In a PIM, the process moves to the data and may perform computations on the components where the data is locally available, transporting only the information needed for coordination within the PIM. While there are many remaining research issues to be addressed, we believe that PIMs offer an important and novel technique for the control of distributed systems.*

Process-integrated mechanisms (PIMs) offer a new approach to the problem of coordinating the activity of physically distributed systems or devices. This includes situations where these systems must achieve a high degree of coordination in order to deal with complex and changing goals within dynamically changing, uncertain, and possibly hostile environments.

Current approaches to coordination all have well-recognized strengths and weaknesses. Centralized coordination uses a single coordinating authority that directs and coordinates the activities of all devices (for example, Stephens and Merx [1990]). This approach, however, incurs a high communication overhead in many domains because the coordinating authority needs to have a complete and up-to-date global view of the world as well as information about the operational state of each of the devices. In addition, the overall system is inherently fragile, as any damage to the controlling authority can render the

Copyright © 2010, Association for the Advancement of Artificial Intelligence. All rights reserved. ISSN 0738-4602

entire collection leaderless. While election mechanisms exist to select a new leader, recovering the state of the distributed system after the failure of the leader is often complicated. The chief advantage of having a single controlling authority, however, is simplicity of implementation and predictability of overall coordinated behavior.

Distributed coordination, on the other hand, involves each device making decisions about its own actions and maintaining its own world view. As a result, such systems can be more robust to component failure. To be effective as an overall system, each device requires something akin to social negotiation with the other devices, with all its concomitant uncertainties and communication costs. Distributed devices cannot make effective global decisions about what information to communicate to all involved parties to enable effective decision making without a global view of the world. Many systems have been developed that are intended to demonstrate such distributed coordination, such as Teamcore (Tambe 1997), CAST (Yen et al. 2001), and BITE (Kaminka and Frenkel 2005), but they remain most effective for preplanned, relatively loosely coupled distributed activities. Tightly coupled coordination that requires near real-time adjustment of the behavior of multiple agents remains a significant challenge for such approaches. In addition, the overall behavior of distributed intelligent systems is typically very hard to predict, making them problematic for many applications.

Partly as a reaction to these problems, biologically inspired approaches attempt to avoid explicit coordination altogether. In some of these approaches, organized behavior emerges dynamically from the collective actions of swarms of simple devices. These approaches have proven difficult to program for predictable behavior and do not have the capability for highly coordinated activity or for quickly changing group focus as the situation changes. Another variant on this theme is architectures where devices do not negotiate but broadcast their current activities and each device decides its own course of action individually (for example, Alliance [Parker 1998]). While leading to robust behavior, coordination is necessarily only loosely coupled.

Each of the aforementioned approaches has applications where they are effective. We propose a novel architecture to add to the mix, a process-integrated mechanism that enjoys the advantages of having a single controlling authority while avoiding the structural difficulties that have traditionally led to its rejection in many complex settings. In many situations, PIMs improve on previous models with regard to coordination, security, ease of software development, robustness and communication overhead. In the PIM architecture, the components are conceived as parts of a single mechanism, even when they are physically separated and operate asynchronously. As you will see, a PIM model bears many resemblances to the Borg in *Star Trek*—a set of highly autonomous agents that somehow share a single "hive mind." A PIM is a mechanism integrated at the *software level* rather than by physical connection. It maintains a single unified world view, and the behavior of the whole mechanism is controlled by a single coordinating process, but this process does not reside on any one device.

The PIM Model

The core idea of the PIM is to retain the perspective of the single controlling authority but abandon the notion that it must have a fixed location in the system. Instead, the computational state of the coordinating process is rapidly moved among the component parts of the PIM. More precisely, a PIM consists of a single *coordinating process* (CP) and a set of *components* each capable of running the CP. The CP cycles among the components at a speed that is sufficient to meet the overall coordination needs of the PIM, so that it can react to new events in a timely manner. The time that the CP runs on a component is called its *residency time*. Each component maintains the code for the CP, so the controlling process can move from component to component by passing only a small run-time state using the mechanisms of *strong mobility* (Suri et al. 2000). The underlying PIM run-time system manages the actual movement of the CP across the components and presents the programmer with a virtual machine (VM) in which there is a single coordinating process operating with a unified global view although, in fact, the data and computation remain distributed across the components.

The PIM model addresses some key problems associated with traditional approaches. In comparison to centralized approaches, the PIM model ameliorates the robustness problem because the coordinating process is not resident (except transitorily, with backups at all other nodes) at any one location. It also ameliorates the communication bottleneck by moving the process to the data rather than the data to the process. While multiagent systems (MASs) address the robustness issue, they introduce significant complexity and uncertainty in achieving highly coordinated behavior. The PIM approach removes the complications of negotiation protocols and timing problems in distributed systems, enabling an ease of programming and a conceptual simplicity similar to centralized models and often offering conceptual advantages over the centralized model.

Time-Sharing	Process-Integrated Mechanism
One processor, many processes	One process, many processors
Supports the illusion that each process is the only one on the machine	Supports the illusion that there is one entity controlled by a single process
Each process sees only its "allocated" memory (a subpart of the overall system's memory).	All memory on every processor is accessible as though it were one memory.
Processes must be switched frequently enough to maintain the illusion that each is running all the time.	Process must execute on each component frequently enough to maintain illusion that it can control any of the components at any time
Programmer writes program as though it is the only one on the machine.	Programmer writes program as though there is a single machine.

Table 1. Comparing the PIM Model with Time-Sharing

The PIM Illusion

The PIM model presents the programmer with an intuitive abstraction that greatly simplifies controlling distributed coordination. One can draw an analogy to time-sharing. Time-sharing models revolutionized computing because they allowed multiple processes to run on the same computer at the same time as though each was the only process running on that machine. The programmer could construct the program with no concern for the details of the process switching that is actually happening on the processor. To the programmer it is as though the program has the entire processor, even though in reality it is only running in bursts as it is switched in and out. The PIM model, on the other hand, provides a different illusion. To the programmer the PIM appears as one process running on a single machine, but the CP is actually cycling from component to component. Furthermore, the programmer sees a single uniform memory, even though memory is distributed, like in a distributed memory system. In other words, the set of components appears to be a single entity (that is, a PIM). The programmer need not be concerned with the details of moving the CP among the components or on which component data is actually stored.

In the time-sharing model, it is important that each process be run sufficiently frequently to preserve the illusion that it is constantly running on the machine. Likewise, with the PIM model, it is important that the CP run on each component sufficiently frequently so that the PIM can react appropriately to any changing circumstances in the environment. The contrast between the time-sharing model and the PIM model is shown in table 1.

An Example

To help make this concrete, consider some example PIM systems. First, consider a trivial PIM system involving fine-grained coordination. Say we have two robotic manipulators, R1 and R2, that must act together to lift a piano (see figure 1). Each arm has a sensor, H, which indicates the absolute height of its "hand," and an action, *Adjust-Arm,* which raises or lowers its arm by a specified distance. There are inaccuracies in the actuators so the actual height levels have to be continually monitored. We present a simplistic algorithm to control the robots so that they lift the piano while keeping it balanced: *if the difference in height of the arms is greater than some value e then adjust R2's arm to match R1's. Otherwise, raise R1's arm a small increment.* We omit the required termination condition in this simple example.

```
if |R1:H - R2:H|> e
        Then R2:Adjust-Arm(R1:H - R2:H)
        Else R1:Adjust-Arm(.2)
```

The prefixes indicate the "namespace" for each arm. Note that this program is for illustration only—we do not claim that this is a particularly good solution or that other methods cannot solve this problem more effectively. The point here is to provide a simple, concrete example to illustrate the idea.

This example illustrates some key points about PIMs. First, as mentioned above, the code for the CP does not involve any explicit intercomponent communication to obtain information or coordinate behavior. Second, neither arm has independent goals or needs to reason about the goals of the other arm, even though the arms' behavior makes it appear that they are sensing and adjusting to each other. These arms are *not* independent enti-

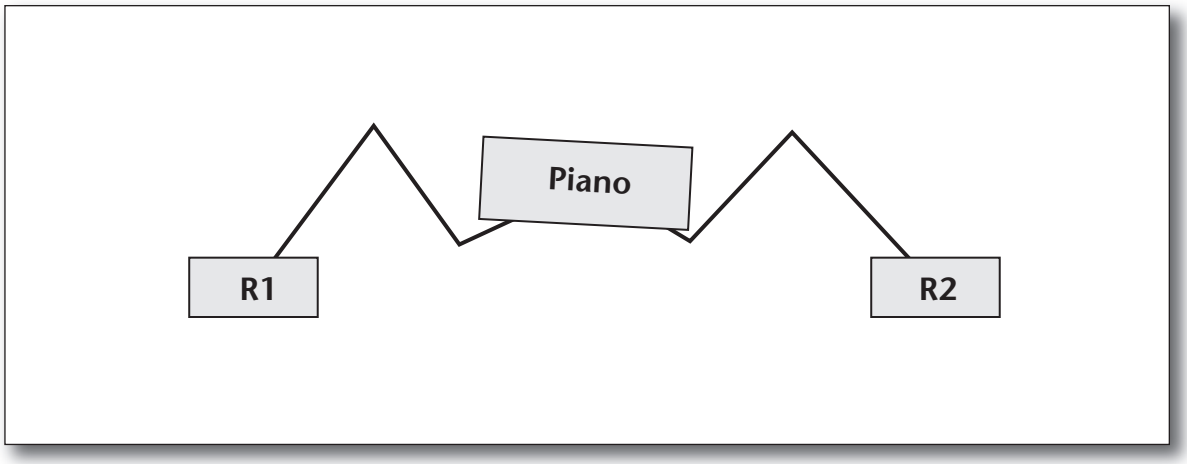


Figure 1: A Simple Coordination Problem.

	Execution Possibility No. 1		Execution Possibility No. 2		Execution Possibility No. 3
R1	Evaluate expression R1:H - R2:H Cache value R1:H	R1	Evaluate R1:H - R2:H Cache value R1:H	R1	Evaluate R1:H - R2:H Cache value R1:H
R2	Evaluate R1:H - R2:H Cache value R2:H Cache result of evaluating the expression (say .1), Execute R2:Adjust-Arm(.1)	R2	Evaluate R1:H - R2 Cache value R2:H Cache result of evaluating the expression (say .1)	R2	Evaluate R1:H - R2:H Cache value R2:H
		R1	Nothing	R1	Evaluate R1:H - R2:H Cache result of evaluating the expression (say .1)
		R2	Execute R2:Adjust-Arm(.1)	R2	Execute R2:Adjust-Arm(.1)

Table 2. Three Possible Executions of R2:Adjust-Arm(R1:H - R2:H)

ties. Rather they act like two parts of a single entity, just like our two arms act as part of our body. The only difference in this case is that these arms are physically disconnected while our arms are physically attached through our body. A key insight underlying the PIM idea is that physical contiguity has no computational significance.

While this article is concerned with the conceptual framework of PIMs, just for once we will drill down a level and consider the actual execution of the CP in the PIM model. Table 2 shows three possible ways the code fragment *R2:Adjust-Arm(R1:H - R2:H)* executes. These differ solely in terms of where the CP is resident during each step of the computation. Note that to be effective, process shifting must occur quickly enough relative to the robots' required rate of movement to keep the piano balanced. As long as this is true, the programmer does not care which of these possible executions occurs. This is just as in tthe componentime sharing, where the actual number of times your program is executing is irrelevant to the computation as long as the process runs quickly enough.

As a second example, consider a loosely coupled set of intelligent components, each with considerable autonomy yet needing to coordinate high-level goals, for example, a RoboCup team, in which each robot does all its own motion planning, can control the ball, and negotiates coordination with the other agents. In contrast, in a PIM version of this scenario, the previously independent robots become components of the PIM (they joined the Borg). They can still perform reactive functions locally, such as controlling the ball. However, the CP plans which roles each robot component plays and coordinates behaviors such as passing. In planning a pass, the CP computation is simple as it can set the time and location of the pass and know where the receiver will be. In other words, each robot component has no need to try to predict the behavior of the other robot components because they are each part of the same mechanism.

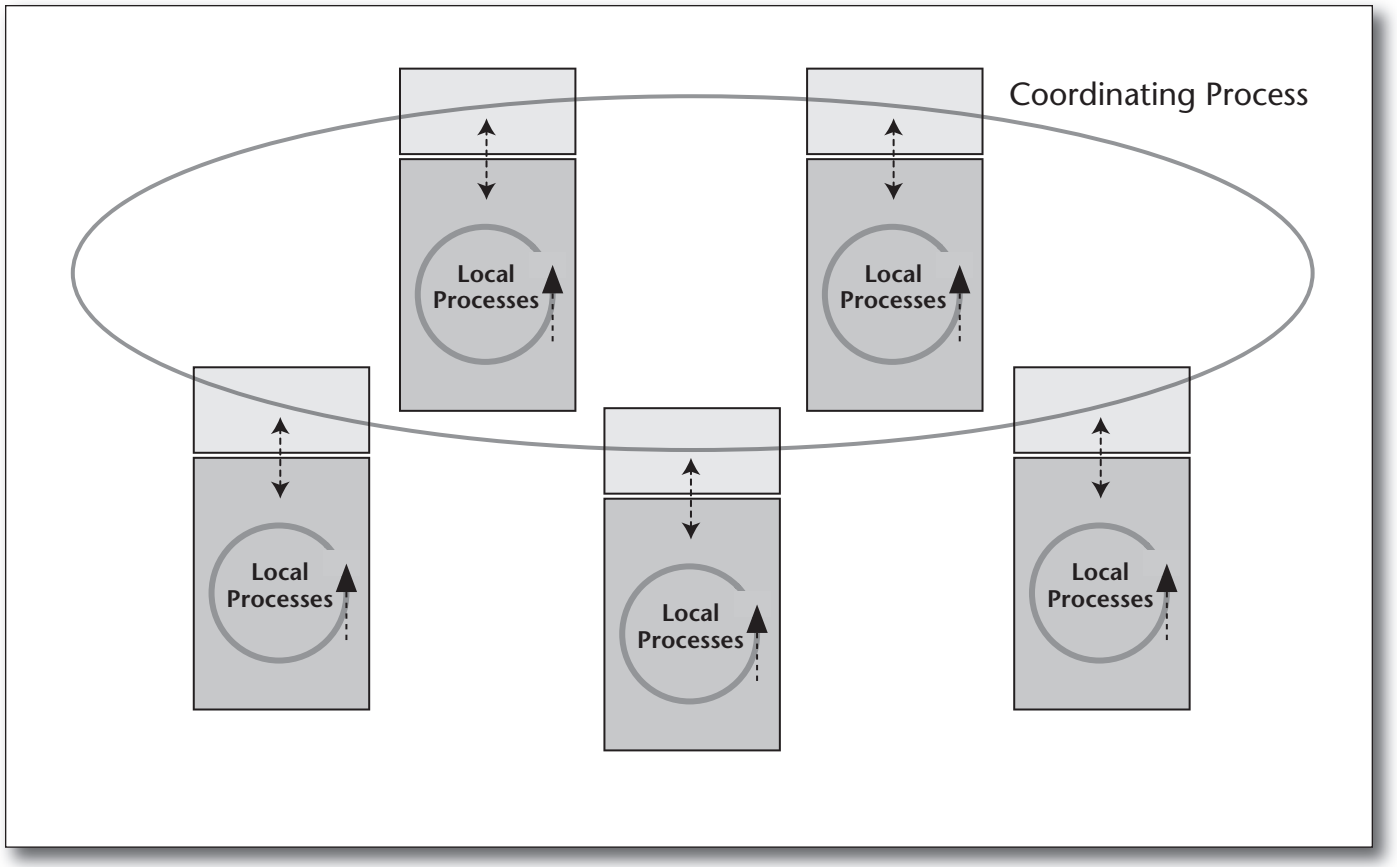


Figure 2: Local and Global Processing among Components in a PIM

A Component-Based View of a PIM

So far we have focused on the programmer's perspective, that is, from the point of view of someone who is trying to solve a distributed coordination problem. We've suggested that PIM offers a way to enable the programmer to describe a solution that abstracts away the details related to intercomponent communication. In this section, we will consider a PIM from the perspective of one of its components, where the overall system is shown in figure 2. Each component has an arbitrary set of local processes running that perform local computations. Local processes are any processes that can run without coordinating their activity with another component. In a robot, these would include processes that interpret sensor data, processes that control the actuators, as well as local planning processes that connect the component's current goals to its activity. In addition, the component is sometimes running the CP that coordinates with other components. To the component, it appears that the CP is always running and has a global view of the situation and "omniscience" about the behavior of the other components.

It is important to clarify what a component is not. It is not an independent agent that makes its own high-level decisions and coordinates its activity with other agent components. There is no negotiation or any other explicit communication between components and no notion of independent goals. A component is much like one's hand—it receives "action directives" from a single decision-making process (the brain) and executes them. It does not have to reason about coordinating with the other hand as the brain is coordinating the activities of both. However, this analogy is still very approximate, as in a PIM there is no single physical part that acts as a "brain," but there is a single computational process that coordinates its behavior. An external observer of the PIM-controlled system might infer that each component is an independent agent in the sense of something that executes its own sense-think-act algorithm independently. This is another aspect of the PIM illusion.

Analyzing the PIM Model

It might seem that it is simply too expensive to be moving the CP rapidly among the components. It turns out, however, that the amount of informa-

No. of Components	Cycle Time 400 ms	800 ms	1200 ms	1600 ms
4	75%	88%	92%	94%
8	50%	75%	83%	86%
12	25%	63%	75%	81%
16	N/A	50%	67%	75%
20	N/A	38%	58%	69%
24	N/A	25%	50%	63%

Table 3. Percent Effective Computation Time Given Cycle Time and Number of Components.

(Assuming 25 ms communication cost per hop.)

tion that needs to be transferred between components can be quite small. First, all the code for the CP is resident on each component, so only the execution state needs to be transferred. At the minimum this would be the current process stack—the stack in the virtual machine with sufficient information so that the next step in the process can be executed. Beyond that there is a time-space trade-off on how much data is transferred with the process. At one extreme, no data is transferred and computations that involve a memory access might block until the CP is once again resident on the component that physically manages that memory location. More optimized performance can be obtained by moving a cache with the coordinating process so that many delays are avoided.

Table 3 shows the trade-off between the reactivity of the PIM and the amount of computation it can do. A *longer* residency time reduces the total fraction of time lost to transmission delays, thereby increasing the computational resources available to the CP algorithms. This increases the latency of the CP as it moves among the components, thereby decreasing the coordination and reactivity of the PIM. Conversely, a *shorter* residency time enhances the system's ability to coordinate overall responses to new and unexpected events since the overall cycle time of the CP will be shorter. But as we reduce the residency time, we increase the ratio of the overhead associated with moving the CP and thus decrease the computation available to the CP for problem solving. In the extreme case, this could lead to a new form of thrashing, where little computation relevant to coordination is possible because most cycles are being used to migrate the CP.

These competing factors can be characterized by the following formulas:

Cycle-time = #components * (Residency-Time + Time-to-move-CP)

Percent-computation-available = Residency-Time / (Residency-Time + Time-to-move-CP)

There is a clearly a trade-off between reactivity (that is, cycle time), number of components, and the percent effective computation available. As the number of components grows, or the required cycle time diminishes, the CP gets less processing time. Note that with the speed of current processors, and the fact that the CP needs mainly to do management and decision functions and can offload expensive computation to local processors, even modest percentages may be quite adequate. However, when designing a PIM for an application that requires very fast coordinated reactions, one may need to limit the number of components in order to attain the needed reactivity.

Note that this trade-off could be explicitly monitored and balanced during execution. When faced with the sudden need for increased coordination, the CP might temporarily decommission some components, decreasing the cycle time among the remaining components without reducing the residency time.

Note also that if the perception and response to an event are wholly local to one component, then the component may react independently of the CP, much like "reflex" responses in animals. It is only those responses that require the coordination of multiple components that are sensitive to the cycle time. A PIM that has a very rapid cycle time can realize highly coordinated behavior, like an athlete, but has relatively little time to compute a "thoughtful" response. A PIM that has a relatively slow cycle time has more computation time for reflective thinking at the expense of rapid responses, like a thoughtful academic.

Before we consider another example, consider one more comparison. The execution of a PIM can be compared to a single central processor model, where the cycling of the CP in the PIM model corresponds to a cycling of queries for sensor updates from each component. This captures the observation that when the CP is resident on a component, it has access to the most recent data on that com-

	Seers	Trackers	Herders
Symbol	S	T	H
Type	UGV	UAV	UGV
Sensor	Shape	Heat	Shape and Heat
Sensor Range	30m	50m	15m
Speed	2 m/s	167 m/s	4 m/s

Table 4. The Robot Characteristics.

	Sheep	Cats	Rocks
Symbol	*s*	*c*	*r*
Temperature	warm	warm	cold
Size	medium	small	medium

Table 5. Objects in World.

ponent. The centralized model can access the same data with a query to the component, at the expense of a high communication overhead. The difference between these two models is that in the PIM model, the coordinating process moves to the data, where in the centralized model the data moves to the coordinating process. In a large number of modern applications involving sensors, it is the data that is overwhelmingly large, so much so that the thought of communicating it all to a single processor is untenable. PIM models function perfectly well in such situations, since the executing process moves to the data rather than the data to the process. Unlike the centralized model, a PIM always "knows" everything known at all of its components within a single CP cycle.

A More Detailed Example: Sensor Fusion

Let us consider a more extensive example, one that we have implemented in a simulated environment. The problem is an elaboration of the pursuit domain (Benda, Jagannathan, and Dodhiawala 1986) and involves three types of robots herding sheep in a simple world that contains two other types of objects, rocks and cats. The types of robots are shown in table 4. The *Seers* are ground vehicles with shape sensors having a range of 30 meters. Based on shape, *Seers* cannot distinguish between animals and rocks, but can tell large (sheep or rock) from small (cat or rock). The *Trackers* are aerial vehicles and have infrared sensors, so they can distinguish animals from rocks, but cannot distinguish cats from sheep. Finally, the *Herders* are ground vehicles that have both sensors but only a very limited range. The properties of the objects in the world are shown in table 5.

To simplify the example for this article, let us assume that all sensors are totally reliable and that each robot has accurate GPS-based positioning so the location of each is known (and stored locally on each robot). The more complicated cases can be handled by PIM models (in fact the more complex the problem the more compelling the PIM model is), but we need a simple case for expository purposes. Given these robots to control, the task is to herd a set of sheep into the center and keep them there, as shown in figure 3.

We will focus on that part of the coordinating process that is doing the sensor fusion to identify sheep. To simplify the example, let us assume the Seers and Trackers follow predetermined paths specified in the coordinating process code. It is often useful in designing a PIM program first to solve the problem assuming an omniscient, centralized processor, that is, a centralized processor that knows everything that is known to any component in the PIM. Given these assumptions, we can implement the global view using a grid/array representation of the world indicating the presence of sheep at each coordinate location, where

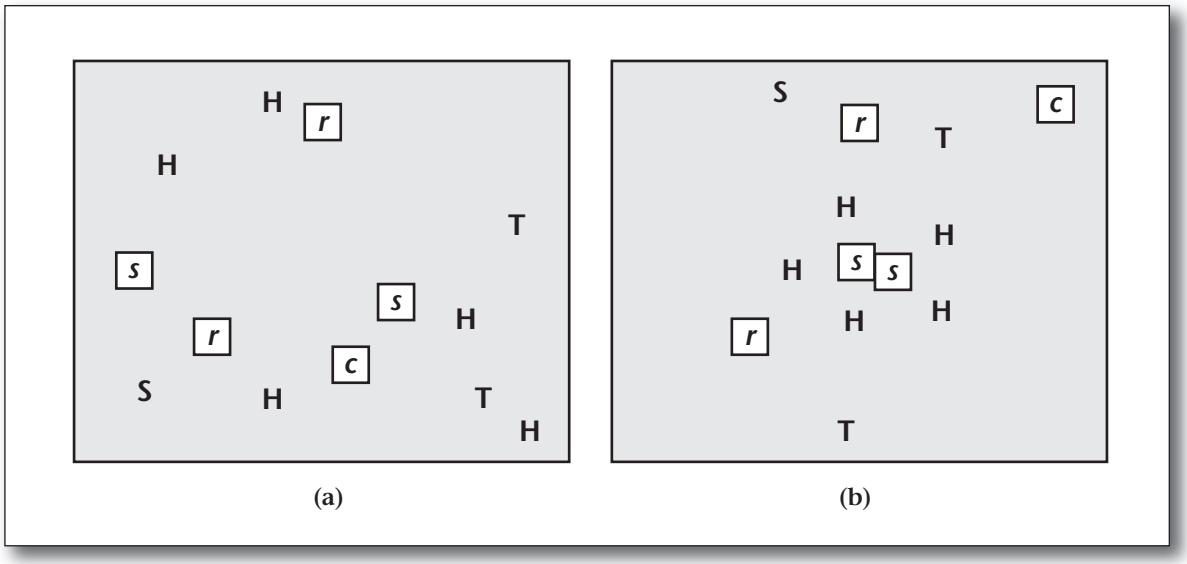


Figure 3. Herding Sheep.

(a) Initial World: Sheep Scattered. (b) Final World: Sheep Rounded Up.

New Observation at (*x, y*)	Old Value	Updated Value
None	Anything	None
Warm	Sheep-shape	Sheep
Warm	Sheep	Sheep
Warm	Warm or none	Warm
Sheep-shape	Warm	Sheep
Sheep-shape	Sheep	Sheep
Sheep-shape	Sheep-shape or none	Sheep-Shape
Sheep	Anything	Sheep

Table 6. Updating the World State.

the values might be (sheep, sheep-shape, warm-object, or none). A new observation consists of a feature value, its coordinates, and the time of observation. Each new observation can be used to update the global view using the information summarized in table 6. Because of the nature of the data, an efficient implementation would use a data structure suitable for representing sparse arrays.

With this, the algorithm for the omniscient centralized process would be:

> Loop {When observation (*f, x, y*) arrives,
> Update global view entry for
> position (*x, y*) according to table 6}

Now we can consider how to implement this on the PIM. The first observation to make is that virtually the same algorithm could run directly on the PIM! The only change would be to use a priority queue based on time-stamp order. Each component would locally process its sensor data and queue a set of new observations. When the coordinating process is resident it has access to the new observations and processes them as usual. In the simplest implementation, we would move the global view data (as a sparse matrix representation) with the process. If this creates a coordinating process that is too heavyweight for the application, there are many ways to reduce the size of the data stored by trading off execution time (see figure 4). We could, for instance, distribute the global view evenly across all the components and not move any of it with the coordinating process. We could then define some fixed cache size of new observations that is moved with the coordinating process. When the CP is resident on a component that stores the relevant parts of the global view for some of the observations, they are removed from the cache and replaced with new observations. In the

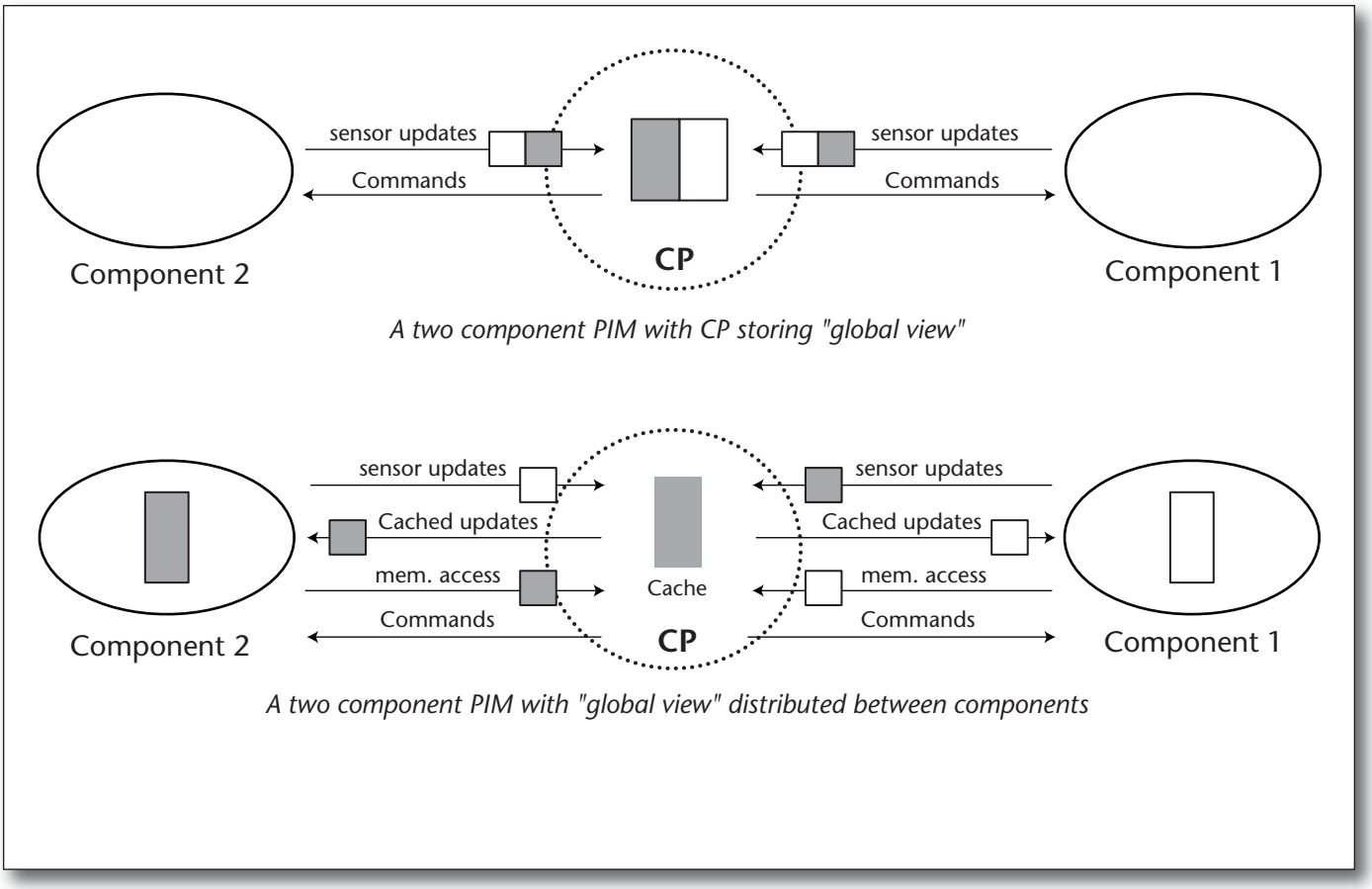


Figure 4. Ways of Maintaining a Global World View.

worst case assuming a computation speed sufficient to process all new observations, an observation would stay in the cache for at most one cycle through the components. By adjusting the cache size, we can trade off effective computational speed for size of the CP footprint. This trade-off is similar in many ways to the trade-off between effective computation speed and page swapping with virtual memory.

As this example illustrates, there are some key trade-offs between moving the global view in the CP versus storing it in a distributed fashion in the component memories. The best approach will depend on the information and the criticality of the data. Note in the distributed version, if a component is destroyed, its part of the global view will be lost and will need to be recomputed from new sources. On the other hand, if the global view is moved with the CP, then the CP has to be lost before the PIM loses data. In this case, as we discuss in the next section, we can revert to a previous version of the CP stored on a component that is still running. In the sensor fusion case above where the world is changing dynamically based on new sensor data, distributing the global view does not incur much risk (as it would have been recomputed soon anyway).

Note that while the actual CP algorithms resemble those for a centralized system with a global view, the PIM version of this code would be much simpler. To implement these algorithms with a true centralized process, much of the code and processing would be concerned with collecting and managing the global view. This would entail either significant communication overhead to transmit all the data to the central process or complex code in the centralized process to determine when to query the other components to obtain new sensor data.

Robustness to Failure

One of the key advantages of the PIM is the robustness to component failure. PIMs can produce robust behavior in the face of component failure with little effort required from the programmer.

There is one key requirement on the style of programming the CP in order to attain significant flexibility and robustness, namely that the CP should not be written in a way that depends on being resident on any specific component. Rather, it should

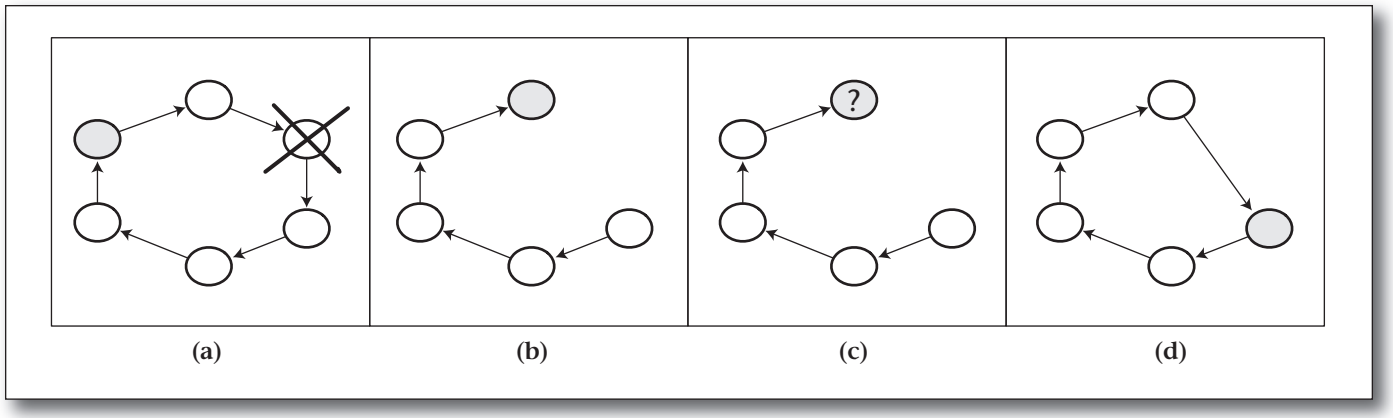


Figure 5. Loss of a Component.

Shaded component shows the coordinating process. (a) The component is lost. (b) CP moves to next process. (c) Problem is detected. (d) CP is passed to next component in cycle.

be written in terms of available capabilities, maintained by the PIM run time. For example, with the sheep herding, the CP cares about what herding robots are available and where they are but does not care about the identity of specific components. The algorithm continually optimizes the activities of the herders that are currently available. If a particular herder component is destroyed, then the PIM run time may be able to "recruit" a new herder invisibly to the CP. Likewise, whenever a new herder becomes available (that is, added to the PIM), the CP can then utilize the new herder in the next cycle.

With this requirement at the algorithmic level, the CP can handle the loss or gain of new components transparently. There are two possible situations that can occur when a component is disabled (or loses communication capability with the other components). The situations differ only in whether the CP is resident on the component at the time it is destroyed. In the case where a component that does not have the resident CP disappears, the PIM run time will detect that the component is missing at the time it attempts to move the CP to it. In this case, the CP is forwarded on to the next component in line in the cycle, and the list of available components is updated. If there is a possibility that the component is only temporarily out of communication range, the run time might continue to poll for it for some amount of time before it is considered permanently lost. The process is shown in figure 5.

The case in which a component is destroyed while the CP is resident is slightly more complicated. The CP is lost and will not be passed on. It is fairly simple to have a time-out mechanism in the run time so that this situation is detected. In that case, the run time then restarts using a copy of the CP from that last known active component. As the CP migration restarts, the PIM continues as before. Because of the short cycle time of the process, the CP is only slightly out of date, and the PIM continues probably without any noticeable effect (except that whatever happened on the component that was destroyed is now missing). This process is shown in figure 6.

Note that we do lose some recent state if the CP needs to be recovered, but it is only slightly out of date. In many domains, this loss is insignificant, especially since the lost computation only involved new data from the lost component. Whether losing a small amount of data is important depends on the domain. For example, a RoboCup team of robots constantly needs to be recomputing its state in every cycle anyway, so losing the most recent computation does not matter much. In domains with critical actions that should only be performed once (for example, withdrawing funds from an ATM), if the knowledge that this action was just performed was lost, then we would need some way to infer that it had been done from the current state (for example, checking whether your bank balance is lower). Domains with critical events with no observable effects would not be amenable to our recovery strategy, but it does not seem that the other approaches would do better in this situation.

More complex cases arise when communication links fail, which could lead to two separate subgroups each operating with its own CP unaware of the other. And if communication links are reestablished, we might have two CPs operating on the same cluster of components. There are relatively simple strategies for detecting an obsolete CP and disabling it simply by not passing it on. We are still studying the full range of complications in order to prove robustness properties as well as an upper bound on the time taken to recover from various types of failure.

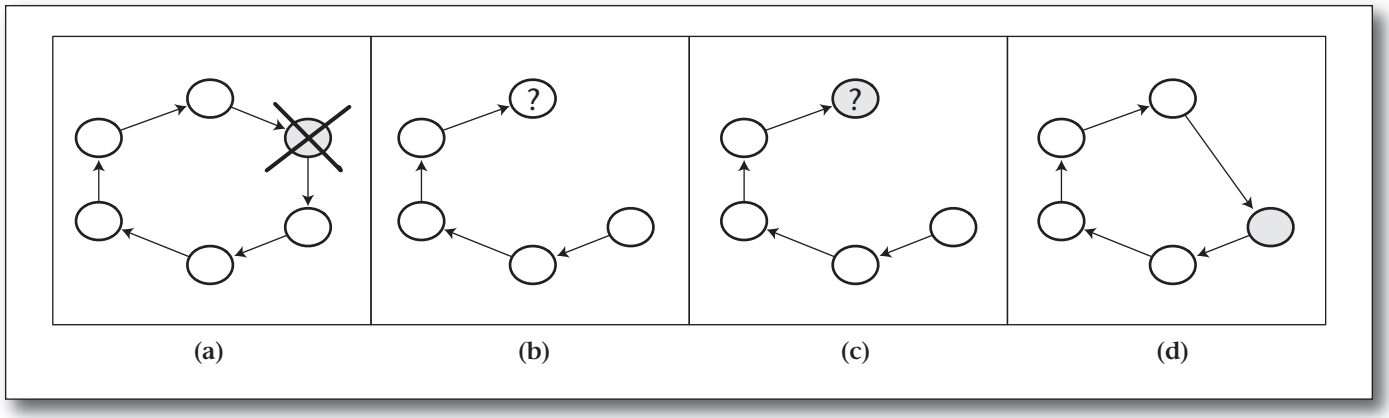


Figure 6. Loss of a Component with Coordinating Process.

(a) The component and CP are lost. (b) Problem is detected. (c) Old copy of CP activated. (d) CP is passed to next component in cycle.

Some Initial Evaluations

There is clearly both more theoretical and practical work required to validate some of the claims we have made. But we hope we have convinced the reader that the approach has promise. Here we provide some preliminary experiments we have performed to explore the framework.

We have developed two different prototype implementations of the PIM. Both implementations support the use of Java as the programming language for the CP. The first implementation uses the Aroma Virtual Machine (Suri et al. 2001) whereas the second version uses a modified version of the IBM Jikes Research Virtual Machine (Quitadamo, Cabri, and Leonardi 2006). Both of these VMs provide the key technical capability for realizing the PIM run time—the ability to capture the execution state of Java threads that are running inside the VM. This allows the PIM run time to asynchronously stop the execution of the CP, capture the state of the CP, migrate it to another node, and restart the execution of the CP. This entire process is completely transparent to the CP itself.

Each component in a PIM has a PIM run time, which is the container for the VM that executes the CP and provides the capabilities of detecting other nodes, establishing network connections, detecting node failure, and migrating the CP from node to node. The PIM run time also provides any necessary interfaces to the underlying hardware. In the case of Aroma, the PIM run time is implemented in C++, whereas in the case of Jikes, the PIM run time is implemented as a combination of C++ and Java.

Figure 7 shows an implemented architecture using the Aroma-based PIM run time, with the necessary components to communicate with a robotic platform. In this particular implementation, the robotic hardware consists of Pioneer robots from Mobile Robots. The PIM run time also provides hardware interfaces to a GPS receiver and an indoor positioning system, along with other utility libraries.

MRLib (the mobile robotics library) provides a rich API for the CP to interact with the underlying robotic hardware. In particular, MRLib provides methods to read sensors and move the robot with commands that do waypoint navigation, with or without collision avoidance. The type of processing done at the level of MRLib is an example of the local processing that can occur at each node independent of the CP and in parallel with other nodes. When the CP is resident on a particular node, it interacts with the MRLib instance on that node.

To demonstrate feasibility, we measured the performance of the prototype implementations of the PIM run time. In the first experiment we used three laptops connected with a high-speed network. A simple CP that performs a matrix multiplication of a 100 x 100 matrix was used, and the residency time was set to 50 ms. The results show that the per hop migration cost is quite reasonable, on the order of 5 – 10 ms. The second experiment used wireless connections and measured the impact of the size of the state information in the CP with respect to migration times. Three laptops were connected using an 802.11b-based ad hoc network operating at 11 Mbps. The CP was modified to carry a variable-sized payload. The residency time in this test was set to 100 ms. The experiment measured the round-trip times for payloads of size 0, 1024, 10240, and 20480 bytes. The results for the Jikes implementation are shown in table 7.

To evaluate our claims that PIM models simplify the code complexity compared to a multiagent approach, we implemented a simplified version of the Capture the Flag game, played by two teams with two to seven players on each side. One team

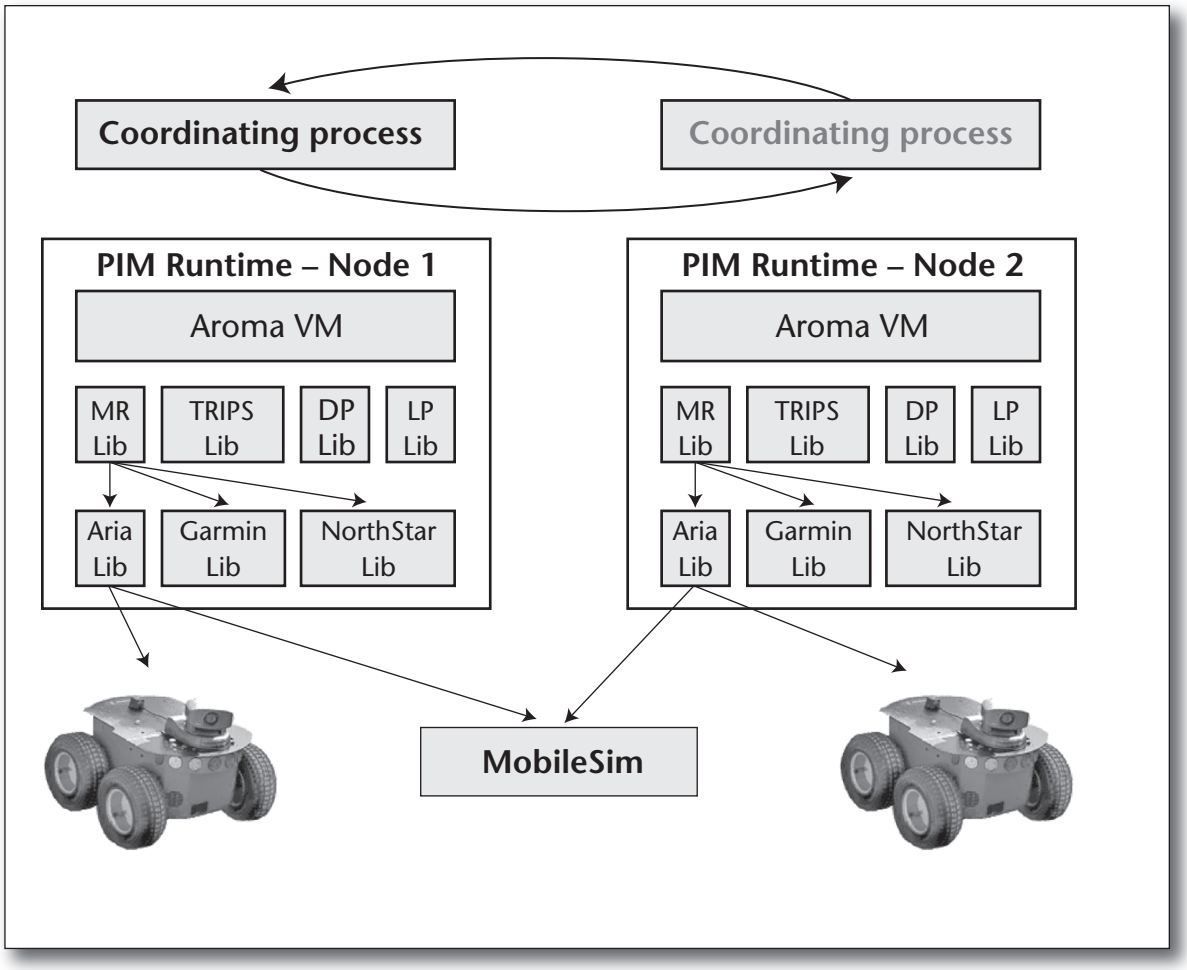


Figure 7. Architecture of the PIM Run Time on Robotic Hardware.

Jikes VM: Payload Size (bytes)	Average Cycle Time (ms)	Overhead per Hop (ms)
0	330.85	10.28
1024	334.95	11.65
10240	381.47	27.16
20480	446.76	48.92

Table 7. Migration Performance Using 11 Mbps Network.

is a PIM, whereas the other team is a multiagent system. The agent-based system is implemented using the A-globe multiagent platform (Šišlák et al. 2005). A-globe is a lightweight multiagent platform that supports mobility and integration with powerful environment simulation components. A-globe has been used successfully for free-flight collision avoidance among autonomous aerial vehicles and also for modeling collaborative underwater mine-sweeping search. The environment simulation components support realistic testing of the multiagent algorithms and facilitate straightforward migration to real distributed environments (such as robotic hardware).

The detailed results of this experiment are reported in other papers (Ford et al. 2008), and we

	PIM		Multiagent	
Component	Lines	Classes	Lines	Classes
Coordination	671	11	4851	33
Role Assignment	538	3	1027	7
Path Planning	430	3	681	2

Table 8. Code Complexity Comparison between PIM and Multiagent Systems.

only summarize the results briefly here. As a measure of code complexity, we counted the number of classes and the lines of code for each of the two solutions. While this has weaknesses as a measure of complexity, it is still widely used. The solution had three primary components: a role assignment component (that decided whether a robot would be an attacker or a defender), a path planning component, and the main coordination component. Table 8 shows the results of the comparison.

As the results show, there is a significant difference in the number of classes and the lines of code required for the Coordination component. Of course, this is just one comparison against a single framework. To make strong claims about code complexity we would need to perform similar analyses with implementations in other frameworks, such as the teamwork models.

Discussion

In this section we will address a number of issues that often are sources of confusion in understanding the PIM model. Many of these points have been raised before, but it is good to revisit them now that we have discussed the ideas in more depth. As we do this, we will compare the approach to other approaches in the literature.

The first key point is that a PIM is *not* a mechanism for controlling a set of autonomous agents. The components do not negotiate with each other (for example, Contract Nets or other approaches [Smith 1980; Aknine, Pinson, and Shakun 2004]) and thus avoid issues of communication complexity from the need to negotiate (Endriss and Maudett 2005). The PIM model does not place constraints on the internal workings of a component except that it prohibits components from explicitly communicating with each other to coordinate behavior. Under PIM control, a set of components may resemble and act like a set of agents to an outside observer, but there is no communication between agents that resembles negotiation of behavior. As such, PIM models resemble a centralized approach to coordination as described by Stone and Veloso (2000):

> Centralized systems have a single agent which makes all the decisions, while the others act as remote slaves... A single-agent system might still have multiple entities—several actuators, or even several physically separated components. However, if each entity sends its perceptions to and receives its actions from a single central process, then there is only a single agent: the central process. The central agent models all of the entities as a single "self."

Note some important distinctions to make. First, in a PIM there is no component acting as the central controller, although the PIM does have a single process, namely the coordinating process that migrates between the components. Second, there is no need for a component to "send its perceptions" to this process, as the process migrates to it. Taking into account these differences, our notion of component is quite similar to the above description.

Our approach for taking the process to the data rather than the data to the process has similarities to the smart-messages approach (Borcea et al. 2002) for distributed embedded systems. Borcea et al. are interested in controlling networks of embedded systems (NESs) in which availability of nodes may vary greatly over time. A smart message is a code and data combination that migrates through nodes that can provide good execution environments for the process (for example, availability of sensor data locally). They argue that smart messages provide a robust mechanism for computation in volatile environments and avoid the need to move large amounts of data. We claim similar advantages for the PIM model.

Smart messages are organized around specific computations that need to be completed and are designed to not depend or care on which node the computation is done. The PIM likewise does not care on which component the computation is done, but the focus of the computation is on coordination of the current set of components. That is why the behavior of a PIM model resembles a mul-

tiagent system while the actual computation resembles a centralized coordination of actuators. In the smart-messages approach, the size of the code that needs to be passed by caching code at nodes is reduced. In the PIM models, we take this one step further: the entire CP code is preloaded on a component when it is initiated, so we never need to pass code, just the current execution state. Another key difference is that smart messages do not follow any specific routing pattern but move opportunistically between nodes. The PIM CP follows a systematic migration pattern as it is focused on coordinating the real-time behavior of all the components and must visit each component frequently enough to meet the reactivity needs of the coordinated system. And one final important difference: the code in smart messages must explicitly invoke the mechanisms for migration—in a PIM the CP code does not need to consider migration issues as this is handled by the PIM run time. In fact, migration is invisible to the CP.

Scalability

A critical limitation on the PIM model appears to be the number of components a single PIM can support. As the number of components grows, the overall cycle time of the CP increases (or the residency time decreases, limiting CP computation). As such, PIM models are more appropriate for systems involving tens to hundreds of components rather than thousands. However, in most domains involving large numbers of components, highly coordinated activity is unlikely to be needed. Rather, the components would probably cluster into groups that must be coordinated at, say, the second level, while the groups themselves can be more loosely coupled (for example, groups perform relatively independent activities possibly with some coordination checkpoints). In these applications, we can foresee hierarchical PIM models, with one PIM controlling each subgroup, and then higher-level PIMs that coordinate the subgroups. This is not an unusual organization. In fact, all effective large human teams that must coordinate activity (for example, companies, the military) use the same hierarchical organization to effectively control team behavior.

Comparison with Multiagent Systems

Stone and Veloso (2000) identify a number of advantages of multiagent systems over single-agent (centralized) systems, including speed-up from parallel computation, robustness to failure, modularity leading to simpler programming, and scalability. It is worth considering how these issues fare in the PIM model. There is ample opportunity in a PIM to exploit parallel computation using the processors on its various components. We have already discussed the robustness issue and claim that a PIM can offer significant advantages over a MAS where close coordination is required. When a MAS loses a key agent, reconfiguring the remaining agents through a negotiation process could be very complex. We also demonstrated above that PIMs offer a much simpler programming model, relieving the programmers of the need to develop negotiation protocols and strategies for sharing data. With regard to scalability, there is an inherent limit to how many agents or components are viable in applications where there is a need for highly reactive coordinated responses. With a PIM the limit is imposed by the required short cycle time, while with a MAS, the limit is imposed by the required negotiation time. In applications involving loosely coupled behavior with little need for fast coordinated reactivity, both approaches can support large numbers of components/agents.

PIMs have an advantage over MASs in applications that require a capability to rapidly to refocus the activity of the entire set of components/agents. In a MAS, a potentially complex negotiation must occur to change the goals of each agent. The PIM model, on the other hand, shares the advantages of a centralized approach where one process makes the decision for all, and the team can be refocused in a single cycle of the CP. Furthermore, MASs have difficulty maintaining a global view. The PIM model inherently maintains a global view of the situation. Finally, a core problem with complex MAS systems is the difficulty in reliably describing and predicting system behavior, even when the agents are only loosely coupled.

Summary

We have described a new model for distributed computation that we believe offers significant advantages in many situations. The PIM model offers promise as an effective infrastructure for handling tasks that require a high degree of time-sensitive coordination between the components, as well as a clean mechanism for coordinating the high-level goals of loosely coupled systems. PIM models enable coordination without the fragility and high communication overhead of centralized control, but also without the uncertainty associated with the system-level behavior of a MAS.

The PIM model provides an ease of programming with advantages over both multiagent systems and centralized architectures. It has the robustness of a multiagent system without the significant complexity and overhead required for interagent communication and negotiation. In contrast to centralized approaches, it does not require managing the large amounts of data that the coordinating process needs to compute a global view. In a PIM, the process moves to the data and may perform computations on the compo-

nents where the data is locally available, sharing only the information needed for coordination of the other components. While there are many remaining research issues to be addressed, we believe that PIMs offer an important and novel technique for the control of distributed systems.

References

Aknine, S.; Pinson, S.; and Shakun, M. F. 2004. An Extended Multi-Agent Negotiation Protocol. *Journal of Autonomous Agents and Multi-Agent Systems* 8(1): 5–45.

Benda, M.; Jagannathan, V.; and Dodhiawala, R. 1986. On Optimal Cooperation of Knowledge Sources—An Empirical Investigation. Technical Report BCS–G2010-28, Boeing Advanced Technology Center, Boeing Computing Services, Seattle, WA.

Borcea, C.; Iyer, D.; Kang, P.; Saxena, A.; and Iftode, L. 2002. Cooperative Computing for Distributed Embedded Systems. In *Proceedings of the 22nd International Conference on Distributed Computing Systems* (ICDCS), 227–236. Los Alamitos: IEEE Computer Society.

Endriss, U., and Maudet, N. 2005. On the Communication Complexity of Multilateral Trading: Extended Report. *Journal of Autonomous Agents and Multi-Agent Systems* 11(1):91–107.

Ford, K.; Suri, N.; Kosnar, K.; Jisl, P.; Pěchouček, M.; and Preucil, L. 2008. A Game-Based Approach to Comparing Different Coordination Mechanisms. In *Proceedings of the 2008 IEEE International Conference on Distributed Human-Machine Systems*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kaminka, G., and Frenkel Flexible Teamwork in Behavior-Based Robots. In *Proceedings of the 20th National Conference on Artifical Intelligence* (AAAI 2005), 108–113. Menlo Park: AAAI Press.

Parker, L. 1998. ALLIANCE: An Architecture for Fault Tolerant Multi-Robot Cooperation. *IEEE Transactions on Robotics and Automation* 14(2): 220–240.

Quitadamo, R.; Cabri, G.; and Leonardi, L. 2006. Enabling Java Mobile Computing on the IBM Jikes Research Virtual Machine. In *Proceedings of the 4th International Conference on the Principles and Practice of Programming in Java 2006* (PPPJ 2006), 62–71. New York: ACM.

Šišlák, D.; Rehák, M.; Pěchouček, M.; Rollo, M.; and Pavlíček, D. 2005. A-globe: Agent Development Platform with Inaccessibility and Mobility Support. In *Software Agent-Based Applications, Platforms, and Development Kits*, ed. R. Unland, M. Klusch, and M. Calisti, 21–46. Basel: Birkhauser Verlag.

Smith, R. G. 1980. The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver. *IEEE Transactions on Computers* C-29(12):1104–1113.

Stephens, L., and Merx, M. 1990. The Effect of Agent Control Strategy on the Performance of a DAI Pursuit Problem. Paper presented at the 10th International Workshop on Distributed Artificial Intelligence, Bandera, Texas, October 1990.

Stone, P., and Veloso, M. 2000. Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots* 8(3): 345–383.

Suri, N.; Bradshaw, J.; Breedy, M.; Groth, P.; Hill, G.; and Jeffers, R. 2000. Strong Mobility and Fine-Grained Resource Control in NOMADS. In *Proceedings of the 2nd International Symposium on Agents Systems and Applications and the 4th International Symposium on Mobile Agents* (ASA/MA 2000), 2–14. Berlin: Springer-Verlag.

Suri, N.; Bradshaw, J.; Breedy, M.; Groth, P.; Hill, G.; and Saavedra, R. 2001. State Capture and Resource Control for Java: The Design and Implementation of the Aroma Virtual Machine. Paper presented at the 2001 USENIX JVM 01 Conference Work in Progress Session (Extended version available as a Technical Report), Monterey, California, April 23–24.

Tambe, M. 1997. Towards Flexible Teamwork. *Journal of Artificial Intelligence Research* 7:83–124.

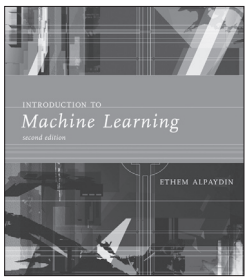
Yen, J.; Yin; Ioerger; Miller; Xu; Volz CAST: Collaborative Agents for Simulating Teamwork. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence* (IJCAI 20011135–1144. Menlo Park: AAAI Press.

Kenneth M. Ford is founder and CEO of the Institute for Human and Machine Cognition (IHMC)—a not-for-profit research institute with headquarters in Pensacola, Florida, and a new laboratory in Ocala, Florida. Ford is the author or coauthor of hundreds of scientific papers and six books. Ford's research interests include artificial intelligence, cognitive science, human-centered computing, and entrepreneurship in government and academia. He received a Ph.D. in computer science from Tulane University. He is emeritus editor-in-chief of AAAI/MIT Press and has been involved in the editing of several journals. Ford is a Fellow of the Association for the Advancement of Artificial Intelligence (AAAI), a member of the American Association for the Advancement of Science, a member of the Association for Computing Machinery (ACM), a member of the IEEE Computer Society, and a member of the National Association of Scholars. Ford has received many awards and honors including the Doctor Honoris Causas from the University of Bordeaux in 2005 and the 2008 Robert S. Englemore Memorial Award for his work in artificial intelligence. Ford served on the National Science Board (NSB) from 2002–2008 and on the Air Force Science Advisory Board from 2005–2009. He is also a member of the NASA Advisory Council and currently serves as its chairman.

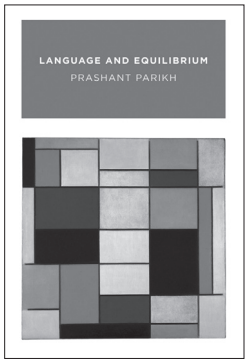
James Allen is an international leader in the areas of natural language understanding and collaborative human-machine interaction, the John H. Dessauer Professor of Computer Science at the University of Rochester, and the associate director and senior research scientist at the Institute for Human and Machine Cognition. Allen's research interests span a range of issues covering natural language understanding, discourse, knowledge representation, commonsense reasoning, and planning, especially the overlap between natural language understanding and reasoning. Allen is a Fellow of the Association for the Advancement of Artificial Intelligence and former editor-in-chief of *Computational Linguistics*. He received his Ph.D. from the University of Toronto.

Niranjan Suri is a research scientist at the Institute of Human and Machine Cognition. He received his Ph.D. in

The MIT Press



Introduction to Machine Learning
SECOND EDITION
Ethem Alpaydın
"A comprehensive exposition of the kinds of modeling and prediction problems addressed by machine learning, as well as an overview of the most common families of paradigms, algorithms, and techniques in the field. The volume will be particularly useful to the newcomer eager to quickly get a grasp of the elements that compose this relatively new and rapidly evolving field."
— Joaquin Quiñonero-Candela, coeditor, *Data-Set Shift in Machine Learning*
584 pp., 172 figures, $55 cloth



Language and Equilibrium
Prashant Parikh
"The book is an intriguing mixture of linguistics, computer science, game theory, and philosophy. It does much to illuminate an enduring mystery: how language acquires meaning." — Eric S. Maskin, 2007 Nobel Laureate in Economics
360 pp., 45 illus., $40 cloth

Now in Paper
Semi-Supervised Learning
edited by Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien
A comprehensive review of an area of machine learning that deals with the use of unlabeled data in classification problems: state-of-the-art algorithms, a taxonomy of the field, applications, benchmark experiments, and directions for future research.
528 pp., 98 illus., $26 paper

To order call 800-405-1619 • **http://mitpress.mit.edu**
Visit our e-books store: **http://mitpress-ebooks.mit.edu**

computer science from Lancaster University, England, and his M.Sc. and B.Sc. in computer science from the University of West Florida, Pensacola, Forida. His current research activity is focused on the notion of agile computing, which supports the opportunistic discovery and exploitation of resources in highly dynamic networked environments. He also works on process-integrated mechanisms—a novel approach to coordinating the behavior of multiple robotic, satellite, and human platforms. Suri has been a principal investigator of numerous research projects sponsored by the U.S. Army Research Laboratory (ARL), the U.S. Air Force Research Laboratory (AFRL), the Defense Advanced Research Projects Agency (DARPA), the Office of Naval Research (ONR), and the National Science Foundation (NSF). He has authored or coauthored more than 50 papers, has been on the technical program committees of several international conferences, and has been a reviewer for NSF as well as several international journals.

Patrick J. Hayes is a senior research scientist at the Institute of Human and Machine Cognition. He has a B.A. in mathematics from Cambridge University and a Ph.D. in artificial intelligence from Edinburgh. He has been a professor of computer science at the University of Essex and philosophy at the University of Illinois, and the Luce Professor of cognitive science at the University of Rochester. He has been a visiting scholar at Université de Genève and the Center for Advanced Study in the Behavioral Studies at Stanford and has directed applied AI research at Xerox-PARC, SRI, and Schlumberger, Inc. At various times, Hayes has been secretary of AISB, chairman and trustee of IJCAI, associate editor of *Artificial Intelligence*, a governor of the Cognitive Science Society, and president of AAAI. His research interests include knowledge representation and automatic reasoning, especially the representation of space and time; the semantic web; ontology design; image description; and the philosophical foundations of AI and computer science. During the past decade Hayes has been active in the Semantic Web initiative, largely as an invited member of the W3C Working Groups responsible for the RDF, OWL, and SPARQL standards. He is a member of the Web Science Trust and of OASIS, where he works on the development of ontology standards. Hayes is a charter Fellow of AAAI and of the Cognitive Science Society.

Robert A. Morris is a computer science researcher in the planning and scheduling group in the Intelligent Systems division at NASA Ames Research Center. He received a B.A. from the University of Minnesota and a Ph.D. from Indiana University. He has led a number of projects related to observation planning and scheduling for remote sensing instruments. His technical interests include optimization planning and scheduling and representing and reasoning about preferences.