Self-Adaptive Software Composed of Port-Based Agents

Kevin R. Dixon Theodore Q. Pham Pradeep K. Khosla

Institute for Complex Engineered Systems
Carnegie Mellon University
{krd,telamon,pkk}@cs.cmu.edu

Prepared for the *Institute for Complex Engineered Systems Technical Report Series*, 2000.

31 January, 2000

Abstract

To facilitate the design of large-scale, self-adaptive systems, we have developed the Port-Based Adaptable Agent Architecture. This distributed architecture allows systems to be created with the flexibility and modularity required for the rapid construction of software systems that evaluate and modify themselves to improve performance. In this paper, we present the architecture, describe port-based agents, and outline several applications where this flexible architecture has proven useful.

1 Introduction

In the monolithic programming model, increasingly capable systems require increasingly complex software. Multi-agent systems achieve sophisticated capability through complex interactions, not complex software. As such, modularity, reconfigurability, and extensibility are more achievable, and components can largely be tested in isolation. However, most implementations of multi-agent systems do not take advantage of this modularity and reconfigurability because they depend too heavily on the foresight of the author at design time. Reconfiguration is typically a time-consuming manual process that often involves changes to the components themselves. Use of multiple processing nodes further complicates design and reconfiguration. The creation of a general multi-agent software architecture that can learn from its own interactions with the world, evaluate its performance, and adapt itself to achieve better its goals, would find natural use in the distributed system, real-time control, and proxy computing arenas. We propose a distributed system supporting port-based agents as such an architecture.

Analysis of the last five years in computing leads to two key insights. First, the phenomenal growth of computing power, the advancement of miniaturization technologies, and the advent of commodity computers guarantee that computers will permeate every facet of life. Second, the growth of the Internet, fueled by this commodity computing, has redefined what a computer is and how computers are used.

Computers started as large, cumbersome machines which were little more than ballistics calculators. As technology progressed, the mainframe computer was born, which was a calculator that multiple people could use simultaneously via time sharing or batch processing. However, the coupling of networking with the mass production of microprocessor-based computers has shifted the computing paradigm from that of a calculator to the information and control devices that are commonplace today and will become ubiquitous in the near future.

While computer hardware has changed drastically in the past few years, computer software has struggled to keep pace. The centralized, monolithic programming model that was adequate, when treating computers as isolated entities, is poorly suited to distributed, multi-task-oriented computing. For computing to become truly ubiquitous, new distributed, multi-task-oriented programming methodologies must be developed. We believe that distributed, multi-agent technologies offer the capabilities needed. With these notions in mind, we are developing a distributed Port-Based Adaptable Agent Architecture (PB3A) to explore this non-monolithic programming model.

In this paper, related work is described in Section 2, three levels of system adaptability are discussed in Section 3, the overall architecture is detailed in Section 4, the runtime environment is described in Section 5, some applications of the Port-Based Adaptable Agent Architecture are presented in Section 6, and conclusions and future work are laid out in Section 7.

2 Related Work

The ideas underlying Port-Based Agents (PBAs) draw heavily on distributed-systems research, particularly the distributed operating systems research of the last dozen years.

The Sprite operating system, developed by John Ousterhout et al. [3] more than ten years ago, introduced the concept of process migration. Process migration allowed an executing instance of a program to be moved between workstations. The ability allowed a single Sprite user to harness the power of many workstations simultaneously and dynamically.

About the same time as Sprite, a group of researchers at the University of Tokyo created the Galaxy Distributed Operating System [4]. This project emphasized object access-level transparency for all resources. This ability to access system resources uniformly, independent of their locations, further simplified process migration. Recognizing that centralized object naming and locating schemes are inefficient for distributed systems, Galaxy employed distributed schemes to maintain global state. In this system, only processing nodes that require a given resource would cache the naming and locating information of that resource. The port naming and locating mechanisms in the PB3A follow the same general scheme, thereby avoiding the centralized naming bottleneck and vulnerability.

In addition to drawing upon distributed operating systems, the PB3A resembles other mobile object programming environments. One such system is the Emerald Distributed Programming Language [5] developed at the University of Copenhagen. The Emerald project created a distributed programming system for heterogeneous computer networks. This system operates in native code and native data representations on each individual platform, but marshals the data into platform-independent representations on transfer. In order to run an application across multiple platforms, the code for that application must be compiled for each platform. To deal with atomicity differences across various platforms, Emerald code utilizes “bus stops”. An Emerald object may only be migrated from one node to the next when a bus stop is reached. Migrating the object at any other point runs the risk of leaving during a non-atomic operation on one platform that is atomic on another. To ease the programming effort, bus stop generation

is built into Emerald compilers. The PB3A ensures atomicity across heterogeneous platforms by using a similar locking scheme. All PB3A user code runs inside system-maintained locks and cannot migrate until those locks are obtained. The disadvantage of the Emerald system is that the introduction of a new platform may require altering the compilers and runtime libraries on all other platforms. Furthermore, the sharp behavioral differences among platforms may mean that a single application must be rewritten for each platform on which it may run. The PB3A avoids both these disadvantages by using Sun Microsystems Java. Java's representation of code is platform independent and its behavioral specification, with the exception of some graphical user interface-related functionality, is uniform across all platforms for which Java is available.

Other agent-based systems include D'Agents formerly Agent TCL developed at Dartmouth College [2], Odyssey and Telescript from General Magic, TACOMA from University of Tromsø and Cornell University, and Aglets from IBM Research. These environments define agents to be code that can be installed and executed on remote hosts, with the ability to migrate to different hosts during execution. Whereas these systems emphasize the support infrastructure for agent programming paradigms and the communication mechanisms between agents, the PB3A additionally focuses on the internal organization of agents, aims to explore recursively composed systems, and exploits self-adaptivity to cope with changing real-world operating conditions.

The port-based concept is derived primarily from port-based objects, first proposed and implemented by Stewart and Khosla in Chimera [6]. Port-based objects were designed for real-time control applications in a multiprocessor environment with a single high-speed backplane. A link between two objects is created by connecting an output port of one object to a corresponding input port of another object. The informational scope within which the port-based objects exist is a flat, public data structure visible to all objects. This implementation is very efficient for monolithic systems, but it provides no concept of agency (see Wooldridge and Jennings [7]).

The PB3A should be viewed as the natural evolution of the port-based concept. Where port-based objects were designed for multi-processor environments and for direct human-initiated reconfiguration, the PB3A is being designed to utilize loosely coupled distributed computing infrastructures and self-initiated software adaptivity. The modern-day computing paradigm exemplified by distributed and self-adaptive systems absolutely requires the autonomy and self-awareness that are the hallmarks of agent technologies. Software composed from independent, self-aware agents that are able to alter their own structure, are best suited to complete tasks in the case of network latencies, node failures, and general operating condition variations that characterize real-world environments. The PB3A's first advantage over Chimera is that the PB3A uses dynamically loaded Java byte-code to avoid recompiling and relinking of the entire system when new objects are added. Specifically, to support distributed computing, the PB3A augments the notion of the port to include cross-network links, employs an encapsulated memory model to make each PBM self-contained, and utilizes mobile Java byte-code along with the previously mentioned dynamic loading to provide code on demand to individual nodes of the network.

3 Adaptability

As software systems grow in complexity, it becomes infeasible for humans to monitor, manage, and maintain every detail of their operation. From a human-computer interaction standpoint, it is desirable to build systems that can be tasked easily, perform intelligently (as evaluated from the user's perspective), and complete the tasks with little or no human interaction. Recognizing this need, the ultimate goal of the PB3A is to aid in developing systems that are self-adaptive. These systems analyze their performance and can dynamically reconfigure themselves to fit better to the current operating conditions and goals in a distributed environment. From a software perspective, three natural forms of adaptation arise.

The first form of adaptation is parametric fine tuning. Most software is written in terms of algorithms that manipulate data. The behavior of these algorithms depends on their parameters. Much research has been done on estimating the error of the algorithm and using that error metric to modify the parameters. For instance, this could be changing the synaptic weights in an artificial neural network through backpropagation or the coefficients of an adaptive digital filter.

The second form of adaptation is algorithmic change. There is seldom one way to solve a given problem; every different approach to solve a problem or calculate a quantity gives rise to a unique algorithm. Two algorithms designed to address the same problem may behave differently based on the precise circumstances under which they are used. A system that is aware of the current operating conditions and the limitations of the algorithms it employs could dynamically choose and switch algorithms when conditions change. For instance, as lighting conditions vary, swapping a stereo vision algorithm for a HSV-based vision algorithm may improve performance.

The third form of adaptation involves mobility. In a distributed environment, computing resource availability varies by both location and time. Certain nodes may have special-purpose hardware, more abundant memory

and processing power, or lower data-access latency. Software that is aware of the resource conditions under which it operates could migrate to complete its tasks sooner or to make progress in the case of failures.

The PB3A provides the primitives and methodologies by which all three forms of adaptation may be realized and initiated by the software itself when operating conditions warrant.

4 The Architecture

The PB3A is a Java-based programming framework that aims to facilitate the development and deployment of distributed, self-adaptive, multi-agent applications. Unlike sequential programming models that require an application to be a single stream of instructions, the PB3A utilizes a threaded programming model allowing simultaneous streams of instructions. To exploit the power of the PB3A, a solution to a problem must be decomposed into a hierarchy of interconnected tasks. We consider a task to be some flow of execution that takes zero or more inputs, produces zero or more outputs, and may modify some internal state. We call these input and output points "ports", and refer to a fundamental unit of execution as a Port-Based Module (PBM).

In essence, a PBM clearly defines the boundaries, entry points, and exit points of the smallest unit of code in the PB3A. The definition of tasks is recursive; a single task may be composed of multiple, possibly parallel, sub-tasks. To capture this notion, the PB3A allows for the creation of Macros, a special type of PBM that is itself an interconnected collection of PBMs or other Macros. Also, the self-contained nature of the PBM, coupled with its completely specified port mapping dependencies, allows not only for easy distribution and coordination of code modules onto a network of computers, but also for those modules to be mobile. More succinctly, PBMs can migrate from node to node during their execution.

Where the PBM represents the most basic unit of execution, the Port-Based Agent (PBA) represents the most basic unit of self-adaptability. Thus, the self-adapting PBA is the cornerstone of our approach to managing software complexity.

The architecture is logically divided into two halves. On one half, there is the PBM task abstraction, detailed in Section 4.1. Derived from PBMs are three other specialized categories: Macros, Port-Based Agents, and Port-Based Drivers, explained in Sections 4.1.1, 4.1.2, and 4.1.3, respectively. Since these derived categories are based on PBMs, they can be used anywhere PBMs can be used. Furthermore, a PBM may belong to multiple categories. The second half of the architecture concerns the support mechanisms used to maintain the PBM abstraction. These support libraries are referred to as the Runtime Core and are discussed in Section 4.2. Finally, the services that the Runtime Core provides are detailed in Sections 4.2.1 through 4.2.4.

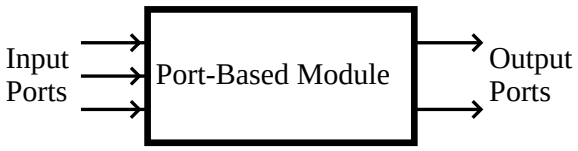

Figure 1: A generic diagram of a Port-Based Module.

4.1 Port-Based Modules

A PBM is similar to Stewart and Khosla's port-based object. Each module has zero or more input ports, zero or more output ports, and possibly some internal state (see Figure 1). All ports are typed in the typical object-oriented programming (OOP) paradigm. A link is created between two PBMs by properly connecting an input port to an output port. Properly connecting an input port to an output port means obeying the OOP rules of inheritance. That is, information that the input port expects must be the same class as, or a super-class of, the information on the connected output port. A configuration can be legal if and only if every input port in the system is connected to at most one output port but output ports may remain unconnected. An output port may map to multiple input ports. PBMs use a localized, or encapsulated, memory model. All state variables specific to an instantiation of a particular PBM, as well as all methods and static members, are contained within the PBM itself. This allows a PBM to be self-governing and independent of other PBMs.

4.1.1 Macros

The first PBM-derived category is the Macro. Since it is common to have repeated subsystems in a given application, groups of PBMs can be grouped together in entities called Macros (see Figure 2). Macros are composed of one or more PBMs and can be defined recursively by encapsulating other Macros. Any ports not connected to other PBMs inside the Macro are external input or output ports of the Macro. Once a Macro is defined, systems can be constructed from the Macro as if it were a PBM. Furthermore, any predefined system can be used as a Macro simply by designating its input and output ports. The PBMs comprising a Macro can run on a single machine, or they can disperse themselves across a network. Thus, a Macro may execute on several computers simultaneously. In this manner, Macros facilitate the development of large-scale systems by providing multiple levels of abstraction and encapsulation.

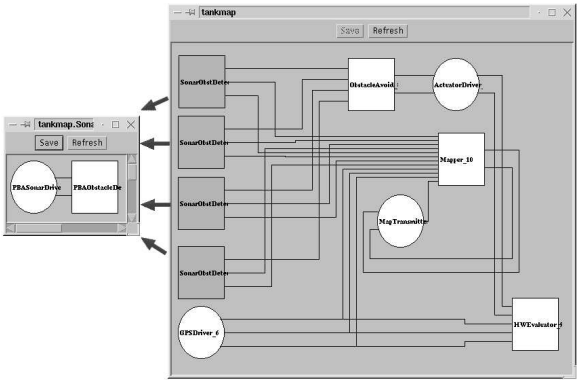

Figure 2: Macros encapsulate one or more PBMs.

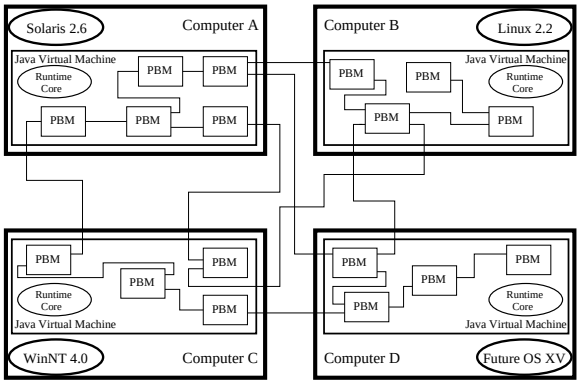

Figure 3: The Runtime Core provides PB3A services.

4.1.2 Port-Based Agents

The next PBM-derived category is the Port-Based Agent (PBA). Where PBMs represent the most basic unit of execution, PBAs represent the most basic unit of self-reconfiguration and self-adaptability. In other words, a PBA is the smallest unit of code that can measure its performance and take steps to improve that performance. These steps may include internal parameter tuning, transferring processing nodes, spawning other PBAs, being replaced by a more suitable PBA, or internal reconfiguration of the PBA itself. In keeping with the modular theme, most PBAs should be Macros of many PBMs. A common PBA Macro configuration consists of one or more PBMs processing data, one or more PBMs measuring the performance of the PBA, and one or more PBMs deciding upon a course of action to improve performance.

4.1.3 Port-Based Drivers

The final derived category is the Port-Based Driver (PBD). Though the PB3A is a high-level programming framework, low-level interactions must be considered. Since most incarnations of Java are inappropriate to interface with hardware, device drivers must be written in a language that can handle pointers, access registers, and other low-level, machine-specific interactions, such as C or C++. A machine-specific driver communicates through a platform-independent "resource port" to a PBD. Presently, resource ports are TCP-based sockets. This allows a PBD to control a specific device from any computer that supports Java, irrespective of where the device is located physically. However, when the device driver and controlling PBD are located on separate machines, latencies can degrade the performance of the device, especially in real-time control applications. For this reason, the PB3A allows any PBM to specify its physical location so that network latencies can be avoided by using a shared-memory model for port information (this will be discussed in Section 4.2). Examples that use PBDs to control devices will be discussed in Sections 6.1 and 6.2.

Furthermore, legacy software also can be interfaced using a PBD. In this case, the legacy application can be considered the device. A PBD can issue commands to the legacy software, and the results can be communicated back to the PBD through the resource port. There is an example of a large-scale system using this methodology in Section 6.3.

4.2 Runtime Core

A traditional computer system is comprised of three logical parts: the hardware that makes up the machine, the core operating system, and the user programs. The hardware offers basic interfaces to computing resources such as CPU, memory, network, and secondary storage. The operating system, or more precisely the kernel, runs on top of this hardware. In most cases, the kernel manages resources and transforms disparate hardware interfaces into a consistent set of services for user programs.

In the PB3A, the Runtime Core is the kernel that runs on top of one or more Java Virtual Machines (JVMs) and provides a consistent set of services to PBMs (Figure 3). The Runtime Core is written in Java and is platform independent. No modifications to the Runtime Core are needed when introducing a new computing environment. In a system involving multiple processing nodes, the Runtime Core instances on each node cooperate to manage the global state of the system. This global state includes the node where a PBM is executing and what port mappings have been established both locally and between nodes. In total, the collective Runtime Core presents a single, consistent computing interface that abstracts the heterogeneous, distributed computing infrastructure.

When a system begins to execute, a per-node daemon starts a JVM and Runtime Core on each of the nodes. Next,

each Runtime Core receives its share of the global system configuration. Each node's share only contains information about the PBMs running locally on that node and any PBMs whose ports are mapped to locally running PBMs. For locally running PBMs, that information includes initial internal state and port mapping data; whereas, for distributed PBMs mapped to locally running PBMs, only port mapping data is included. From this configuration information, the Runtime Cores load all locally executing PBMs. The Java byte-code for those PBMs may come from the local file system or be requested from a network host designated as the code server (see Section 4.2.3). Although all PBMs reside in the shared memory of a JVM on a node, each is self-contained and has no references to the other PBMs, and consequently can have no effect on other PBMs. This memory-space protection is maintained by the JVM. Once all PBMs in the global system have been instantiated, the Runtime Cores begin mapping ports. Ports between PBMs on the same node map directly through shared memory. For mappings between two nodes, the Runtime Cores perform replication (see Section 4.2.1), and the mapping is created using a TCP-based socket. Once all port mappings are established, the Runtime Cores initialize each PBM with its initial state. Finally, a thread is created for each loop-based PBM and the Runtime Cores initialize the per-node event dispatch thread pools (see Section 4.2.2). This completes the distributed initialization phase and subsequently execution begins.

4.2.1 Ports

Current programming models do not allow for easy reconfiguration. The primary difficulties center upon dependencies and hidden interactions. The communication and coordination methods in the current programming models are known only to the sections of cooperating code and are hidden from the rest of the runtime system executing those sections of code. Reconfiguration in such a system requires altering memory addresses and potentially altering the way sections of various communicate. Even multiprogramming and multi-threading environments, where communication rules are more formalized, are not effective.

The PB3A solution to the problem is to formalize communication between cooperating PBMs and rigidly enforce the rules. That is, PBMs can only communicate with each other through type-specific ports. Each PBM is effectively treated as an individual unit that cannot access information in another PBM unless it is explicitly shared. Sharing data requires linking an input port to an output port of like or derived type. This connection results in an information pipeline where the link details are completely hidden in the ports themselves and are available only to the Runtime Core. The indirection and decoupling inherent in the input and output ports allows the Runtime Core to rearrange links arbitrarily without disturbing the PBMs. Consequently, swapping one PBM for another only requires instantiating the new PBM and implanting the ports, and possibly the state of the original PBM, into the new PBM. Some minor bookkeeping information in the ports must be updated, but the PBMs connected to the PBM being swapped need not be disturbed.

The other significant advantage of ports lies in distributed and mobile code. Since the details of all port connections are managed by the Runtime Core, PBMs executing on different processing nodes do not need any special code to communicate. To establish port mappings between PBMs on two processing nodes, the Runtime Core of the input-side node issues a replication request to the Runtime Core of the output-side node if replication of that specific output port has not been previously requested. The output-side node's Runtime Core keeps track of what output ports are replicated to where and sends port state updates to the appropriate nodes every time those ports change value. The consistency of this replication is defined on a point-to-point basis. That is, from sender to receiver, the changes are guaranteed to be delivered sequentially. However, when one output port is replicated to various processing nodes, no guarantees of consistency exist between the receiving nodes. In other words, the two or more receiving nodes are not guaranteed to see the same value at the same time beyond the point-to-point consistency guarantee. This output replication allows PBMs to communicate as if they were running on the same machine, though with potential added latency. Furthermore, port communications hide migration of a PBM between nodes from any PBMs connected to the migrating entity. Hiding port replication and migration free a PB3A programmer to concentrate on the algorithms employed and information communicated, instead of how to communicate. Of course, in some cases the latencies introduced cannot be completely ignored.

4.2.2 Port-Based Module Runtime Structure

The PB3A supports two different, though not mutually exclusive, runtime models for each PBM. A PBM may be threaded, event-driven, or both. A threaded PBM consists of a main method body that is executed within a loop controlled by the Runtime Core. The Runtime Core raises the PBM's lock before executing the method body and releases the lock upon exiting. This guarantees that a threaded PBM will not be migrated while executing its main method body. After each iteration of the loop, and after dropping the PBM lock, the Runtime Core puts the thread to sleep for a PBM-specified amount of time. An event-driven PBM does not possess a thread of its own. Its event dispatching

method body executes only in response to changes on input ports of the event-driven PBM. Before executing a PBM's event dispatching method body, the Runtime Core raises the PBM's lock. Upon completion of event dispatching, the Runtime Core releases the PBM's lock. For PBMs that are both threaded and event-driven, the lock additionally prevents both sections of code from simultaneously executing. The event-dispatching facility itself is composed of multiple threads in a worker pool configuration so that events on different PBMs can execute in parallel.

4.2.3 Loading

As the maintainer of the PBM task abstraction, one of the Runtime Core's responsibilities is instantiating PBMs. The full specification of a PBM instance consists of Java byte-code, internal state, and port mappings. A PBM's internal state and port mapping data, along with its Java class name and PB3A instance name, are referred to collectively as the PBM's configuration. Loading is the process of instantiating a PBM based on its configuration. This process occurs at system startup and in response to requests issued by PBMs. To load a PBM, the Runtime Core must first retrieve the byte-code for the PBM's Java class. The Runtime Core locates this byte-code by first checking the local file system. If the byte-code is available locally, then the Runtime Core invokes the JVM to load the byte-code. Otherwise, the Runtime Core contacts a network host designated as the code repository, the ModuleServer (discussed in Section 5.1), and requests transmission of the byte-code via a network socket. Once the byte-code has been received, the Runtime Core invokes the JVM to load the byte-code. Next, an instance of the PBM class is instantiated and a member method is invoked to set the internal state of the instance to the data stored in the configuration. Finally, the Runtime Core establishes the port mappings of the PBM instance and may allocate a thread if the PBM is threaded.

4.2.4 Migration

PBM migration is the process of transferring an executing PBM from one processing node to another. Migration of a PBM involves capturing its configuration at the source node, transferring that configuration to the destination node, and then reloading the PBM at the destination. Reloading the PBM from a configuration follows the same loading process discussed in the previous section, the primary differences concern capturing the state of an executing PBM and transferring that state across nodes. Just as a PBM has a method to set its internal state from its configuration, a PBM has a method that returns a memory reference graph representing its internal state. When a request is made to migrate a PBM, the Runtime Core raises the PBM's lock and then invokes the internal state graph retrieval method. Raising the lock pauses the PBM and it cannot alter its internal state. The PBM's internal state graph, port mappings, Java class name, and PB3A instance name are then recorded in a configuration record. The Runtime Core of the source node then contacts the Runtime Core at the destination node and transmits the configuration record via a network socket. The transmission of the configuration record is conducted by Java Serialization. Java Serialization is a built-in language mechanism to efficiently convert any memory reference graph into a byte stream. This byte-stream may be a file, a network socket, or a memory buffer. In addition to the configuration, the Runtime Core of the source will transmit any replication requests it has received for the migrating PBM's output ports. The destination Runtime Core then loads the PBM from its activation record and takes control of replication management from the source Runtime Core. The source Runtime Core is then able to deallocate local records and references of the migrated PBM.

5 Runtime Environment

The PB3A runtime environment consists of the ModuleServer, the Launcher, the NetExecutor, and the NetController.

5.1 ModuleServer

The ModuleServer is the server side of the PB3A's client-server code distribution system. Each PBM instance is composed of internal state, port mappings, and code. PBM code is a class (or set of classes) within Java's well-defined package name-space. On client demand, the ModuleServer transmits PBM code to a remote host.

When a client needs PBM code to which it does not have access, the client connects to the ModuleServer. A separate server-side thread handles each connection, allowing multiple clients to make use of the server simultaneously. After connection, the client transmits the fully qualified name of each PBM code class. The ModuleServer resolves the name into a path in its local file system and then transmits the Java byte-code class file to the client. Using built-in Java mechanisms, the client can load the byte-code as a Java class. Then the PBM combines that code, a PBM's internal state, and its port mappings to instantiate the PBM.

New PBM code can be added to the ModuleServer by simply copying their class files into a subdirectory of the ModuleServer's module path. The server does not need to be restarted for this new code to be recognized. Because the ModuleServer never loads the PBM code directly, new versions of existing PBM code can be added by overwriting the old class files. Again, the server does not need to

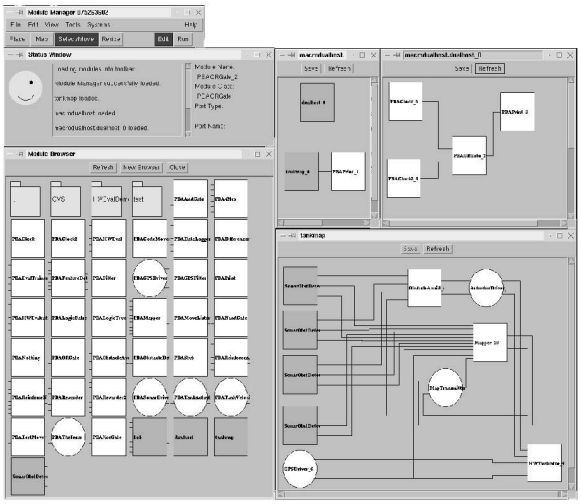

Figure 4: ModuleManager visualizes PB3A systems.

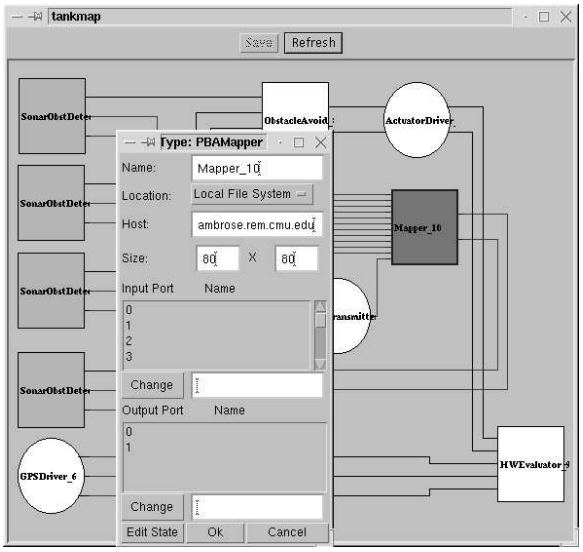

Figure 5: Editing PBM properties via the State Editor.

be restarted. Clients can dynamically clear their locally cached copies of previous PBM code and a new version can be loaded.

5.2 ModuleManager

Inspired by file browsers from graphical operating systems, the ModuleManager is a visual systems configuration tool, shown in Figure 4. Currently, the ModuleManager supports creating and editing system configuration files graphically. To create a new system, the user selects PBMs and then connects their input and output ports to form a configuration by pointing-and-clicking. A file browser-like interface allows users to search the local file system and the ModuleServer for PBMs. Essentially, the ModuleManager is a schematic editor for specifying data pathways in a PB3A system.

The ModuleManager allows the user to set special properties in the PBMs. The "Run Location" specifies on which host machine the PBM should begin execution. If this property is set to "local", then the PBM will execute on the machine that loaded the configuration. If the Run Location is a hostname, the Module will execute on that host. Other useful properties that may be edited in the ModuleManger include the PBM's internal state. This editing can be done by the ModuleManger's built in State Editor via Java Reflection, discussed in more detail in Section 5.3.

When the user has completed a configuration, she may save the configuration to disk or send it directly to the Launcher for execution. The ModuleManager plays no part in this execution.

5.3 State Editor

Most PBMs have internal state and the PB3A allows the editing of this state by the State Editor. When a PBM is selected in the ModuleManager, a set of properties is displayed (Figure 5). These properties include initial host, as well as several other properties, and allows the user to edit the internal state of a specific PBM instantiation. The internal state of a PBM includes variables, objects, or arrays declared outside the scope of any methods in the PBM. This information is obtained by Java Reflection. The State Editor queries the Java Virtual Machine for the names and types of state variables for a particular PBM class. Presently, Java Reflection only returns information on public, protected, or package-default members; information on private members of PBMs cannot be obtained in the State Editor. Next, the State Editor asks the Java Virtual Machine for the current values of the state variables for a specific PBM instance. The State Editor then presents this data graphically to the user, using any object-specific display methods, if necessary. To display primitive types (integers, doubles, etc.) all that is needed is a simple text box. However, some non-primitive objects, such as Bayesian Belief Networks or a Hidden Markov Models, may be displayed in a more sensible, type-specific manner. The State Editor determines if the non-primitive object has a specialized display routine, which is a method written by the author of the object. If such a method exists, then the State Editor uses this method, otherwise the State Editor allows the user to open the object and view its members directly. This process can be repeated recursively through its embedded objects until only primitive types remain.

The author of a PBM does not need to put any hooks or

access routines for the State Editor to gain access to its state variables. This makes the State Editor extremely versatile, and allows the State Editor to manipulate PBMs that have been dynamically added to the PB3A.

5.4 Launcher

The Launcher comprises one of the two system execution entities in the PB3A. It contains a Runtime Core along with code to divide a global configuration into the smaller, node-specific configurations. The Launcher's local copy of the configuration mirrors the global configuration. This makes the Launcher responsible for synchronizing and managing changes to the global configuration. All requests that alter the global configuration are serialized by the Launcher.

During execution, the Launcher has a complete Runtime Core and is responsible for providing a PBM execution environment. The Launcher loads all PBMs from the ModuleServer or local file system, launches threads for threaded PBMs, provides event notification facilities, manages remote port replication, and responds to system reconfiguration requests.

Once the Launcher divides the global configuration into node-specific configurations, it contacts the NetController at each of the remote hosts. The Launcher then requests that NetExecutors be spawned to host that node's piece of the current, global system configuration.

5.5 NetExecutor

The NetExecutor is the second system execution engine in the PB3A. Like the Launcher, the NetExecutor contains a Runtime Core and is responsible for providing a PBM execution environment. Immediately after spawning, the NetExecutor opens a network socket to the Launcher that initiated it. The NetExecutor then uses this network link to download from the Launcher the section of the global system configuration that it must execute. Like the Launcher, the NetExecutor loads all PBMs from the ModuleServer or the local file system, starts threads for threaded PBMs, provides event notification facilities, manages remote port replication, and responds to system reconfiguration requests.

Currently, before NetExecutor can perform any action that would alter its local configuration, it must contact the Launcher and lock the global configuration. Since only one node may be locking the global configuration at a given time, this process guarantees that the NetExecutors provide Launcher with the information needed to serialize all system changes.

The Runtime Cores embedded in NetExecutors and the Launcher cooperate to manage the PB3A port communication system. This local and remote port access transparency frees the PB3A programmer to treat a distributed, heterogeneous network of processing nodes as if it were one large computer.

5.6 NetController

The NetController acts as a gateway daemon through which PBMs may enter a remote processing node. Whenever a NetExecutor or Launcher needs to start PBMs on, or migrate PBMs to, a remote node it must first ensure that a NetExecutor, designated to run PBMs for the current configuration, is running on that node. To ensure this, the NetExecutor or Launcher on the original node opens a network socket to the NetController and transmits the global configuration name. NetControllers listen for network connections on a specific socket number. Upon receiving the global configuration name, the NetController may respond three different ways. If the NetController determines that its node does not have enough computing resources remaining to satisfy the request, it can reject the request. If the NetController determines that it has enough resources and a NetExecutor is already running for the specified global configuration, it replies with the network socket number for that NetExecutor. If the NetController determines that it has enough resources and a NetExecutor is not running for the specified global configuration, the NetController spawns a NetExecutor designated for that global configuration and replies with the network socket number for the spawned NetExecutor. This process ensures that only one NetExecutor is designated for a given global configuration on a given processing node. Moreover, this guarantees that if two PBMs from the same global system configuration are executing on a node, they will be executing in a shared memory space and port communications will work directly through memory references. If a Launcher loses contact with a NetExecutor for some reason, the Launcher can contact the NetController for that NetExecutor's node and request that the NetExecutor be shut down either gracefully or forcefully.

6 Applications

We have used the Port-Based Adaptable Agent Architecture as the substrate for several applications that demonstrate some of the features of the architecture.

6.1 Robotic Mapping

In this scenario, two mobile robots, shown in Figure 6, are tasked to map a laboratory using ultrasonic sonars. One robot is endowed with a simple strategy to explore the laboratory, a Bayesian mapping algorithm, and a hardware monitor to detect any failures that would prevent the robot

Figure 6: Robots used for mapping: Patton and Rommel.

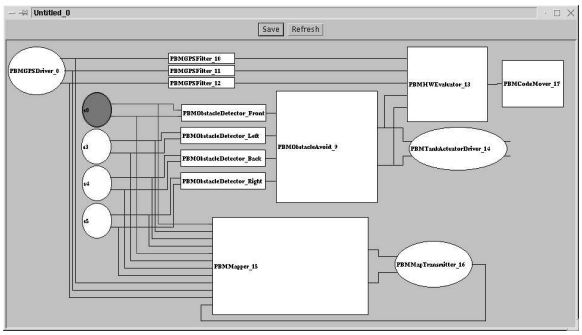

Figure 7: Layout of the PBMs used for mapping.

from completing its task, while the other robot sits idle. If the first robot has a hardware failure (such as a graduate student cutting the power to its motors) then the PBM monitoring the hardware issues a request to the Runtime Core to move the software to the idle robot. The Runtime Core then requests that all PBMs in the system (Figure 7) serialize their state so that any information collected by the first robot can be sent to the second robot. Next, the Runtime Core requests the ModuleServer to send the code for each PBM and the retrieved state information to the second robot. The system finally resumes execution on the healthy second robot. Thus, the second robot can continue mapping after the first robot fails. This transferring of software takes about five seconds once hardware failure is detected.

6.2 Transferring Learned Knowledge

Another application where this architecture has proven very useful is in the transferring of learned knowledge from simulation directly to the real world. For this application, the Port-Based Adaptable Agent Architecture was combined with the mobile robot simulator RAVE, described by Dixon et al. [1]. RAVE provides interfaces and dynamic linking of libraries that allow the same robot code to execute on a simulated or real-world robot.

In this scenario, a mobile robot learns to move to a goal location while avoiding obstacles using a reactive control policy. The robot commences learning in simulation. The simulated robot is endowed with extremely simplified zeroth-order dynamics. Clearly, the dynamics of the real robot are more complex, and the parameters of the controller must be fine-tuned to account for this difference between the real and simulated robots.

When the task is learned satisfactorily in simulation, the Runtime Core moves the entire system from the simulated robot to the real robot. Since RAVE allows the same robot code to execute in simulation or on a real robot, the system can be immediately resumed on the real robot. After a few moments of learning on the real robot, the controller has sufficiently accounted for the difference in dynamics. The transferring of the controller code from the simulated to the real robot takes about two seconds. Learning in simulation, then finishing the learning on the physical system allows for a drastic reduction in total time required to learn the task, while not suffering a decline in performance.

6.3 Traffic Optimization

The PB3A was also used to develop a basic intelligent transportation system: an adaptive, decentralized signal controller for urban traffic. Each intersection has PBMs that collect sensor information, communicate with neighboring intersection controllers or traffic centers, and issue intersection signal commands. Different types of intersection controllers were designed: fixed-time controllers, periodically switching between red, yellow, and green lights; adaptive controllers based on local probabilities and queues; or learning techniques. A performance agent was developed to swap controllers depending on traffic conditions. The PB3A was applied in simulation to a small section of the urban area of the city of Pittsburgh called Penn Circle, as part of a Community Project to improve the traffic in the area. The goal was to study the traffic patterns and current intersection-controller quality compared to more advanced controllers. With the PB3A, it was possible to deploy the whole system very quickly and connect it with the traffic simulator ARTIST developed by Bosch, using a PBD to interface with the simulator. Further developments will include the use of vehicle-to-vehicle communication, inclusion of bus schedules, emergency vehicle activity, and vehicle navigation features.

7 Conclusions and Future Work

We have created a distributed, agent-based architecture that facilitates the development of large-scale, self-adaptive systems. The Port-Based Adaptive Agent Architecture (PB3A) specifies a highly modular and decoupled agent-to-agent communication scheme via input and output ports and provides the necessary primitives for code migration. Furthermore, the PB3A gives some specifications how a Port-Based Agent should be structured, giving agents the autonomy to become truly self-adaptive. Whereas previous research in this area has focused individually on mobility, software composition, or adaptability, we have presented a unified architecture. This unification allows more effective research into self-adaptive systems.

Another powerful notion in the PB3A is the recursive definition of tasks. That is, a PB3A system may be composed of many agents. Each agent may be composed of many tasks which, in turn, may be composed of many subtasks, and so on. The PB3A makes this logical, recursive system realization possible through the encapsulation of tasks in Macros.

The decoupled communication scheme between PB3A modules allows for their testing to be done in isolation. This gives rise to rapid prototyping and reliable systems to be designed and built extremely quickly.

Future work will allow the user to have a more interactive role in a PB3A system execution. Presently, once the system is started, the agents in the system determine when or if the system should adapt itself (moving agents to different nodes, swapping algorithmic modules, etc.). We will allow the user to be able to adapt the system during its execution by specifying any aspect of agents possible. Also, we will modify the State Editor to give the user the ability to view and modify the internal state of a module, in real-time, during its execution. Currently, the user may only use the State Editor to modify the internal state of a module before the system begins execution.

Also, the port and event systems will be made more sophisticated. We will embed environmental information into the ports of a module so that the module can make more informed decisions. Information such as network latencies, network failure rates, and CPU usage could prove useful to self-adaptive agents. Events will be expanded by allowing more types of events, rather than just input port changes, and priorities will be given to different types of events.

Acknowledgments

We would like to thank Andrea Byrnes, Nathan Clark, John Dolan, Enrique Ferreira, Jordan Harrison, Dan Heller, Jonathan Jackson, Rich Malak, Chris Paredis, Yatish Patel, and Charles Tennent for their contributions to this work.

This work was supported in part by DARPA/ETO under contract F30602-96-2-0240 and by the Institute for Complex Engineered Systems at Carnegie Mellon University. We also thank the Intel Corporation for providing part of the computing hardware.

References

[1] Dixon, K.R., J.M. Dolan, W.S. Huang, C.J.J. Paredis, and P.K. Khosla. “RAVE: A Real and Virtual Environment for Multiple Robot Systems”, *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*, 1999.

[2] Gray, Robert. “Agent Tcl: A transportable agent system”, *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management*, December 1995.

[3] Ousterhout, John K., Andrew R. Cherenton, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. “The Sprite Network Operating System”, *IEEE Computer*, v. 21 n. 2, pp. 23 - 36, February 1988.

[4] Sinha, Pradeep K., Mamoru Maekawa, Kentaru Shimizu, Xiaohua Jia, Hyo Ashihara, Naoki Utsunomiya, Kyu S. Park, and Hirohiko Nakano. “The Galaxy Distributed Operating System”, *IEEE Computer*, v. 24 n. 8, pp. 34 - 41, August 1991.

[5] Steensgaard, Bjarne and Eric Jul. “Object and Native Code Thread Mobility Among Heterogeneous Computers”, *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pp. 68 - 78, December 1995.

[6] Stewart, D.B and P.K. Khosla. “The Chimera Methodology: Designing Dynamically Reconfigurable and Reusable Real-Time Software Using Port-Based Objects”, *International Journal of Software Engineering and Knowledge Engineering*, v. 6, n. 2, pp. 249 - 277, June 1996.

[7] Wooldridge, Michael and Nicholas R. Jennings. “Intelligent Agents: Theory and Practice”, *Knowledge Engineering Review*, 1995.