Flexible Internetworking of Devices and Controls

Michelle Munson,† Todd Hodes,‡† Thomas Fischer,† Keung Hae Lee,† Tobin Lehman,† Ben Zhao‡†

† IBM Almaden Research Center
650 Harry Road (K57/D3)
San Jose, CA 95120

‡ Computer Science Division
University of California, Berkeley
Berkeley, CA 94720

Abstract

This paper describes the TSpaces approach to integrated control of heterogeneous devices in heterogeneous control networks. The TSpaces system is an application middleware that provides a uniform layer of indirection between components. The system provides gatewaying between legacy network protocols and IP, asynchronous group communication, a shared data schema, component rendezvous, and event composition. We describe each of these facilities, emphasizing how they enable evolutionary extension of application behaviors, and illustrate with example control applications.

1 Introduction

Control systems for industrial automation have historically followed hierarchical models, organizing devices around a single command controller. Sensors and actuators are commanded by logic controllers, which are themselves physically tied to a centralized software control center, running on a mainframe or workstation. The single controller model suffers from some of the same problems that the PC mainframe world experienced before the explosion of distributed interconnectivity with the Internet, including brittleness to reconfiguration and a centralized failure point. Adding a new device to extend the network requires reprogramming and/or rewiring, and possibly even renumbering the network. Reconfiguring one device in the system may require a top-down reprogramming of the central controller.

Recent trends toward increasingly "smart" and ubiquitous devices make possible a relaxation of the centralized model toward a decentralized, distributed approach. Sensors and actuators are increasingly equipped to communicate peer-to-peer, displacing centralized controllers, much as PCs equipped with more computational power, storage, and memory displaced mainframes. However, limitations in existing approaches to interoperation at the network transport and application levels are constraining the flexible interworking of heterogeneous devices.

Data networking has achieved rapid, global interworking in heterogeneous peer-to-peer systems through the use of a common internetworking protocol, The Internet Protocol (IP). IP enabled the rapid, global interworking of software systems and computer networks by providing a uniform network addressing scheme and routing, atop which various transport protocols — reliable (TCP), unreliable (UDP), and real-time (RTP) — are successfully and widely deployed. More recently, the Java language is creating a common programming environment and network operating system for code mobility, moving toward platform-independent application development. Additionally, emerging software component systems, such as Sun's Enterprise Java Beans/Jini [22], HP e-Speak [10], and OMG CORBA [18], are promoting software re-use and interoperability by exposing interface methods, internal data models, and programming APIs in the form of remote objects.

Until quite recently, these key enabling technologies fueling the Internet have not been practical for controls. Control systems are typically used in real-time systems, on the critical path of a manufacturer's business, for example. While the Java language has many features for reliable programming (e.g., strong typing, required exceptions, no pointers), the first Java solutions for industrial controls failed to achieve deterministic, mission-critical requirements. Early Java solutions were inefficient wrappers for existing soft controllers (designed for mainframes), running on off-the-shelf Java Virtual Machines (JVMs). Inefficiencies in early JVMs, and the complications of optimizing while also maintaining platform independence, made "real-time" Java systems impractical.

Now the rise of customized JVMs for soft real-time and hard real-time [16] is giving way to Java soft controllers and Java Programmable Logic Controllers, PLCs with JVMs on the chip [12]. Additionally, the Java community is standardizing an object model for factory automation that takes into account the critical requirements of controls systems[5]. The reality of real-time Java controls is coupled with an increasing premium on exploiting distributed architectures in control systems, as evidenced by several trends: Smart sensors and actuators are capable of inter-working peer to peer. System extension is commonplace, e.g. adding an additional wing to a building, where a new cooling system is integrated with the main building electrical system, or adding data connectivity to buildings and homes. Manufacturing systems are converging with other enterprise systems, such as electronic commerce portals and purchasing systems.

In short, conditions are ripe for the distributed interconnection of controls. In the sections that follow, we present a strategy for interconnecting heterogeneous devices and controls, using Internet standards and application middleware with which we are experimenting in the TSpaces project. In Section 2 we explain why current proposed networking standards, which assume a vertically-integrated solution, are not sufficiently flexible for internetworking and introduce our approach. Section 3 explains the design concepts of our application middleware; we describe the implementation of these concepts in our system architecture in Section 4. In Section 5 we describe application scenarios using our approach to support the manufacturing processes of an imaginary business. Section 6 discusses future work, and finally, Section 7 summarizes.

2 Motivation

Several vertically-integrated solutions have been presented for standardizing the interconnection of controls and devices for automation [20, 1]. These vertically-integrated solutions define all levels of the communication stack, from the physical link layer, through the network layer, to the application layer [19], fixing compatibility of the two communicating endpoints at all levels. This approach arose at a time when interconnection of heterogeneous endpoints was uncommon; it naturally exploits the top-to-bottom compatibility of the endpoints for optimization of their communication.

The compatibility that the vertically-integrated approaches enforces is too restrictive when the endpoints use heterogeneous protocols (e.g. X-10 controllers talking to BACNet systems). For internetworking, a more flexible strategy is needed. Our approach leverages IP (Internet Protocol) as a common substrate on which to internetwork. We build gateways to translate from heterogeneous protocols to an application level middleware that creates a uniform coordination layer between gateways. The middleware — the TSpaces system — provides a rendezvous for communicating entities, asynchronous group communication, a common interface schema language, and event-based composition.

We have demonstrated the flexibility of this approach by building "remote control" applications for a PDA platform that interfaces to an X-10 controller and media devices such as cameras and video displays; currently we are experimenting with the extensibility of this approach to arbitrary application interfaces.

In our approach, IP provides a uniform transport level protocol. We assume that all control systems to be interconnected will either use IP addressing for all nodes within the system, or that a system using a non-IP networking protocol (e.g. LonWorks, BACNet, X-10), will be interconnected to an IP network via a gateway or bridge. The role of the bridge is twofold: The bridge provides a common IP address for addressing all nodes within the network by routing packets to the proper addresses in its subnet, and the bridge provides basic conversion facilities for translating communication packets from IP to the subnet's internal protocol. A node's address consists of both its gateway address, and an internal address. If the gateway address is not known, an out-of-band lookup of this address, by contacting a service discovery service, may be necessary.

Referring to Figure 1, the TSpaces middleware on IP approach we suggest builds upon the IP "thin waist" to "raise" the level in the communication model where two endpoints must deal with heterogeneity to the application. In turn, the approach adds a degree of flexibility to distributed systems, facilitating rapid, scalable internetworking. When two heterogeneous vertical solutions are connected (Figure 1a) communication between the endpoints undergoes conversions at potentially every level in the communication stack, such as object wrappers, network bridges, and media access controllers. The interoperability characteristic of Internet systems is depicted as an IP thin-waist (Figure 1b). Independent of the physical medium, media access scheme, or applications that ride on

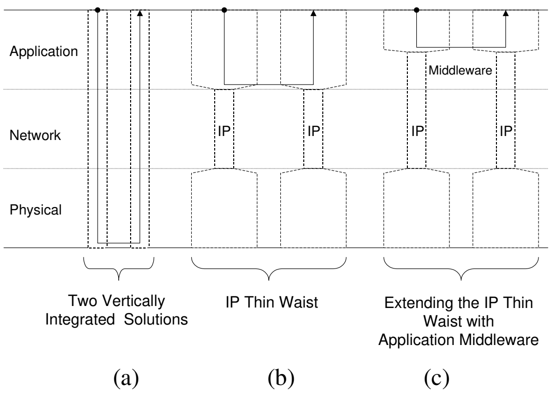

Figure 1: Contrasting a vertically integrated solution, the IP thin-waist, and a middleware thin-waist.

top, IP is a uniform waist through which all communications travel up and down the stack. Our approach (Figure 1c) extends the IP thin-waist by applying an application middleware, the TSpaces system, as a uniform coordination layer, on top of IP. By providing component rendezvous, composition, and an interface schema language and repository, TSpaces raises the level at which application programmers must deal with heterogeneity and provides a uniform layer on which applications for distributed control can evolve.

3 Design Principles

While the vertically-integrated solutions do not offer the flexibility of IP, we recognize their advantages and that not all legacy systems will be replaced with IP-networks and ubiquitous JVMs. Instead, we assume a hybrid model: legacy systems coexist with networks of new Java soft- and hard-controllers and IP devices by talking through protocol gateways which translate between the legacy protocol and IP. Over IP, we apply a common application middleware, the TSpaces system, as a layer of indirection to deal with mismatches in interfacing heterogeneous components. This leads us to the following design principles:

Protocol Gatewaying: "Legacy" controls are interconnected with new network-oriented Java controls through protocol gatewaying and proxies to IP.

Rendezvous: As new devices and services appear and the capabilities of known peers evolve, components must locate the most appropriate peer service or device. The TSpaces Service Discovery Service is designed to work in the rendezvous stage to assist clients in discovering the existence of services; locating an appropriate service, perhaps based on matching complex attributes, such as user methods, metadata, security constraints and QOS attributes; and obtaining an interface for using the service, i.e. discovering the schema of user methods.

Composition: As internetworks are increasingly upgraded, reconfigured, and extended, facilities to compose applications with minimal adaptation — without rewiring, readdressing, or arbitrarily wrapping interfaces — are important. The TSpaces System includes asynchronous group communication using a common data schema (tuples), database facilities for storing/querying interface descriptions, and event composition.

Convergence: The convergence of automation and manufacturing systems with “eBusiness,” business-to-business processes conducted over the Internet, will require easy integration of controls with data systems. IP, combined with the uniform TSpaces system layer, bridges non-discriminately into web servers, traditional I/O devices (printers), and application servers.

Interoperability: As wireless hand-held user devices such as PDAs and other mobile computing devices become more common, mobile clients will serve as “universal appliances” from which users can control any number of devices. The middleware should support wide area interoperability, whereby users and wireless devices in the factory can be moved from one physical location to another without reconfiguration of a running system. A strong separation of interface description from program implementation (using an indirection through middleware) allows the user interface to be re-mapped to new devices as an evolving application interface.

4 The TSpaces Control Architecture

TSpaces [23] is often described as “a network communications buffer with database capabilities.” It allows two connected objects to communicate asynchronously and anonymously via a publish/subscribe communications model. Messages are collections of typed ordered sets called “tuples.” For example, a tuple constructed to tell camera A to rotate left might be:

<“camera A”, “rotate”, “left”>

The basic commands that TSpaces offers are read(), write() and take(), ala the tuplespace concept used in Linda [9, 6]. Reading and taking operate via a “tuple template.” A template for the previous message might look like:

<“camera A”, String.class, String.class>

In this case, the template will match all tuples with “camera A” in the first field and any string in the second and third fields. This template would also match tuples such as:

<“camera A”, “rotate”, “clockwise”>

TSpaces also offers blocking versions of the wait() and take() calls, (waitToRead() and waitToTake()), and offers the capability to register to receive notification when a command is performed on a tuple. To register, the program must provide a command, a template as in a read/take, and a callback object. The callback object will be invoked when a tuple matching the template is acted upon by the specified command.

The primitive TSpace operations form the basis for a middleware system that handles device rendezvous, asynchronous reliable multipoint communication, and event-based control. The use of tuples provides the basis for a universally shared data schema. Together, these allow device control applications using TSpaces to focus on piping together various devices' input/output events rather than the underlying communications. Additionally, the system includes a lightweight downloadable user interface language (in the style of Java applets or HTML forms) that is tied to the tuple-based communication, allowing users with resource-limited client devices to easily communicate with entities using the system.

Applications can choose to use some or all of these mechanisms on a case-by-case basis. Depending on the characteristics of the underlying communication, an application (and thereby devices connected together) may choose any of various protocol for communication (e.g., RMI, TCP sockets, proprietary protocols). Having devices interact directly may be required when the overhead of passing messages through the TSpaces intermediary is prohibitive (for example, mouse movements are likely too bandwidth- and latency-sensitive to be good candidates). In this case, applications still can take advantage of the rendezvous capabilities. The details of our rendezvous facility are presented in Section 4.1.

On the other hand, if applications/devices choose to communicate using a TSpace as an intermediary, they can take advantage of the available asynchronous group communications facilities and existing common data schema. Additionally, applications can leverage the indirection between device endpoints to allow application functions to evolve without changing the endpoint devices' behavior. This flexibility is a key advantage of event-based control through an intermediary. A TSpaces event-driven system decouples the per-device control software from the applications that wish to use it. Devices have only well-defined interfaces describing the kinds of commands they can perform and invoke. To add functionality, a series of specialized intermediaries are added, each one “connecting” the command receptors to the command generators in a certain way to carry out the task that they were assigned to do. As an example, imagine we have an alarm that, when triggered, writes a tuple to the TSpace, and that there is an application registered to receive these alarm events. Upon receipt of such an event, the application writes a tuple that triggers the flashing of light on a system console device. Now, imagine that we add a camera into the system that responds to various “rotate” commands. We could create an agent at the intermediary TSpaces system that instructs the system to listen for the same alarm messages, and upon receiving one, execute a camera “rotate” command to point it in the direction of the intrusion. In this way, the alarm application evolves independent of the endpoint device communications. The details of such event-based composition are described in Section 4.2.

In addition to using the rendezvous and event communications facilities, applications can choose to leverage the TSpace system's Mobile Document Application Language (MoDAL) [8] and its common schema for describing devices' control interfaces. The use of MoDAL UI descriptions and the shared device interface schema allows a user to find and use dynamically discovered devices/controls — thereby enabling universal hand-held “device remote controllers” that interact via user interfaces that are downloaded on-the-fly. The use of downloaded and dynamically generated interfaces in our system is described in more detail in section 4.3.

4.1 Service Discovery

Consider a campus-wide security system built around TSpaces consisting of many different devices and many different device types: card readers, door openers, motion sensors, cameras, etc. To provide the ability to dynamically add and remove components to the system without requiring centralized control, and to provide resiliency to the effects of devices going on-and off-line over time, TSpaces provides a dynamic discovery system. Such a discovery system must provide dynamic advertisement of resources, querying to enable solicitation of these resources, and an infrastructure allowing the system to scale to very large networks. This idea is similar in concept to that of Sun's Jini [22] or the IETF's Service Location Protocol (SLP) [21]. In the context of internetworking of controllable devices, this discovery subsystem is used to allow applications to access various devices based on descriptions of those devices. The exact information that can be provided in a description is flexible; examples include location information, device class/type information, contact information (protocol type, gateway address), etc. Devices register a description of themselves with the system, and users or applications make queries against it, looking for description matches.

To provide these rendezvous capabilities in TSpaces, we extend the approach of [7] for use with tuples rather than XML documents. Tuples are a form of hierarchical, binary, typed values, where the values and hierarchical inclusions may optionally be named. In XML, elements are untyped, must be strings, and must be named. Thus, to leverage the XML-based infrastructure of [7], we need only address these differences. To do so, we represent tuples as markup documents that are an XML-based variant where elements may have empty tags (i.e., "<> ... </>"), values can be encoded in binary (allowing them to contain serialized Java objects),[1] and type information is encoded as element attributes. To allow validation of tuples against a schema in the style of an XML DTD, we represent descriptions of tuples as a form of DTDs called Tuple Validation Fields (TVFs). Currently, TVFs are themselves simply underspecified tuples (identical to the format used for tuple template matching), and thus they have less descriptive power than DTDs. We are currently moving the TVF format from a tuple template to a more general XML-Schema [4] BNF in order to address this limitation.

4.2 Event-based Composition

Applications built using the TSpaces system middleware can take advantage of a distributed event grammar to simplify communication between components. This event-based programming facility is called the Event Exchange; it is a service that can "glue" together the components of a distributed application.

4.2.1 What are Events?

"Events" are loosely defined as named occurrences detected by a software component. Examples include a sensor detecting a physical event or a component signalling an internal state change (i.e., an error condition, a variable changing value, or that a UI widget was clicked). In our model, events are passed between software components as messages; the event messages themselves are typed, as are parameter values of that event. An event is uniquely identified by both the message type and the typed parameter values.

The event-based programming paradigm is a convenient and efficient programming method for "active" systems, where components must frequently respond to state changes in other components [2, 3]. Control networks are populated by hundreds of event sources and receivers (sensors, actuators, ...), and are thus well-matched to the domain.

Most event systems work via a publish-register-notify model ala the CORBA model [18]; a receiving component which has registered an interest in the event is notified of the event's occurrence via a "callback" function. In the callback, a sequential handler routine executes in the receiver.

Events emerged as an efficient alternative to polling in application domains which are driven largely by asynchronous I/O; examples include embedded systems and user-interface programs. Events are an efficient alternative to multi-threaded programming, where they can effectively eliminate some synchronization overhead and thus reduce complexity. The Event Exchange model leverages and extends traditional event-based programming methodology in five key ways: through the use of parameterized event registration to enable fine-grain selection; through the addition of action clauses tied to event notification; through the ability to enable cross-network event handling in addition to intra-process handling; through the ability to build complex sequenced matching criteria based on a grammar (ala regular expressions); and through the ability to "chain" together sets of event operations to enable aggregate behavior. We now describe each of these features, briefly, in turn.

4.2.2 Event Schema

Traditional event programming treats events as single, monolithic types, such as an "alarm" or "mouse_click" event. The parameters of these types are passed to the receiver callback, but cannot be selectively filtered by the event system itself; the receiving application must implement such filtering. The Event Exchange, on the other hand, allows for such fine-grained parameterized registrations. To enable this, it uses an event schema — analogous to an object-schema — where an event is an instance of a class uniquely identified by its type and parameters. Only events matching this registration are notified to the receiver, saving network capacity and sorting/buffering/filtering at the receiver. Additionally, event parameters can be passed from the triggering event occurrences to the action routines that are invoked due to them.

4.2.3 Distributed Events

Traditional event programming assumes that events are communicated from sources to receivers in a single address space. The Event Exchange is targeted at distributed scenarios where events are communicated between component address spaces, across a network. To enable this, event sources publish the event types that they generate to the TSDS as part of their service descriptions. The Event Exchange service acts as a proxy which subscribes for events on behalf of receivers by registering with the tuple space where those events are generated. When an event source generates an event internally, it writes the event to its TSpace server; the Event Exchange then receives a notification, thereby completing the cross-network call.

[1] Binary data can be used in standard XML documents by encoding it, i.e., in Base64 as described in the XML-Data proposal [13]. We use the same approach.

4.2.4 Event Grammar

The Event Exchange provides applications with an event grammar. This grammar is used to specify declarative sequences and combinations of events that must occur, allowing clients to avoid having to manage such possibilities on a per-application basis, ad hoc.

Event grammars are interpreted by the Event Exchange at run-time, allowing the combinations of events to be adjusted dynamically rather than specified statically in component code. Combined with the use of action injection specifications, such "composite" events can replace callback routines.

4.2.5 Composition

The Event Exchange supports a form of application-level "composition" via event-rules that can be chained together through their action clauses. The Event Exchange passes parameter values from notified events into the actions they fire. If the result of an action then generates an event that matches yet another rule that rule will be "chained" to the previous rule, creating a succession of operations. This powerful primitive is similar to continuation passing, data flow, and the "paths" concept of Ninja [17].

We have built a printer conversion service, *PrintAny,* that illustrates this feature. The service takes print requests with files of any format, and attempts to find an appropriate conversion service to convert files to a type supported by a printer, and then print the file. The service needs only one system-level event rule registered with the Event Exchange:

<"printRequest", <String.class file, String.class type>> —>
<"print", <String.class file, String.class type>>

PrintAny works as follows. Clients write printRequest events to local tuple spaces. Converters and printers listen on tuple spaces for print events with parameter type that they support (such as postscript, PDF, etc.). When the rule is registered with the Event Exchange, the Event Exchange queries the TSDS for all clients that source events of type "printRequest" and registers with their local tuple spaces. When a printRequest event is notified to the Event Exchange, it queries the TSDS for services descriptions containing the "print" action. The TSDS query includes the type parameter, so only a converter or printer that supports files of type will be returned. The Event Exchange writes a print event to the local tuple space for that converter or printer. If the receiver is a converter, it converts the file type and writes a printRequest event containing the converted file. This fires the rule again. The Event Exchange is notified and lookups up a receiver that supports the converted type, this time receiving a printer description. The Event Exchange writes a print action to the printer's tuple space. The printer prints the file.

The same service could be implemented by an application that invokes intermediary converters itself; however, such an application would need to be changed whenever a new converter type or a new printer type is added. By instead leveraging the facilities of the Event Exchange through its dynamically interpreted event grammar, the components of the system can evolve, or the system can be deployed in new locations, without recoding.

4.3 Document-Based Interfaces

The final component of the TSpaces system approach to device control is the MoDAL Mobile Document Application Language [8], a technology that allows users with resource-constrained devices to access control interfaces connected through TSpaces. Specifically, MoDAL is a user interface description language and an interpreter which we are developing as a vehicle for the implementation of an "universal information appliance." Based on XML syntax, MoDAL allows a program running on a hand-held device such as an IBM WorkPad or 3Com Pilot to adapt its interface to a specific service. TSpaces is the enabling communication middleware that glues MoDAL programs to actual devices and services on the network.

MoDAL enables a user of a PDA to dynamically discover the control interface[2] of a device, selectively download a user interface that talks to that control interface, and control the device itself by manipulating the user interface elements contained in the UI. Each user interface element in a MoDAL program may be associated with actions to be taken locally or remotely. Remote actions are invoked by a MoDAL client through tuple writes into a TSpace. The device reads the tuples to complete the action.

The common format used by TSpaces for encoding all messages allows the control interface for devices to remain flexible. As all control commands are eventually delivered as tuples, the same interface may serve to control various devices, thereby allowing for the possibility of universal information appliances.

We have built MoDAL/TSpaces programs which control the state of lights connected via X-10 and video conferencing equipment connected via RS-232, and are actively expanding the set of MoDAL clients. In the case of X-10, a user may turn on or off the light with a wirelessly connected IBM WorkPad. A request to turn on the light "N" is represented as <"X-10", "N", "On">. The same interface might also be used to control the opening of a gate "M," whose message would look like <"X-10", "M", "Open">. The utility of this kind of flexibility is discussed in [11].

Our initial experiments with MoDAL confirm that it is a useful, flexible way to control devices. However, most examples we considered so far involve a simple interaction with items such as TV remote controls. We are now developing an architecture that will allow for a more general interaction between MoDAL and devices such as monitoring device status, adapting the interface according to its discovery, and action involving multiple interfaces and control. We have found that it is not obvious how we should handle the more general interaction problems with the thin client approach taken by MoDAL combined with TSpaces — a situation akin to that of the WWW, where HTML forms might be posted to servers running servlets.

A promising avenue for future exploration includes combining MoDAL (and its ability to specify which tuples are generated by particular MoDAL applications) with an event exchange ruleset. This would allow clients to inform the system as to how tuples inserted by particular programs are to be used by the system; this promotes a level of flexibility in solving the heterogeneous interface problem described in [11].

[2] An interface describes the methods a component implements and exports, naming or addressing information describing how to contact a component, and metadata for attributes such as access control.

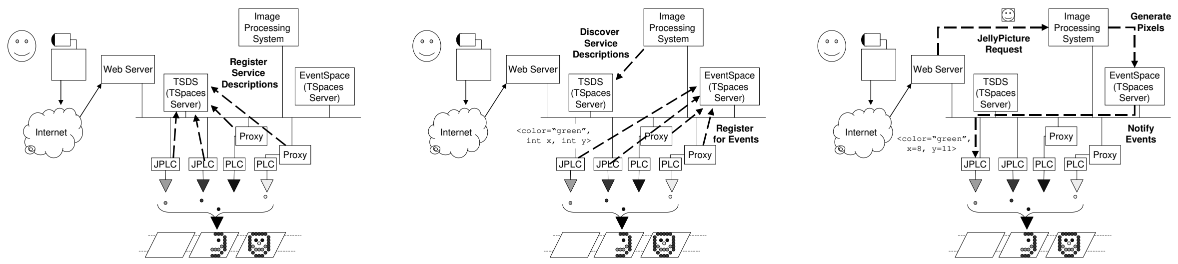

Figure 2: An example of the TSpaces device control system in action. Components illustrated include TSpace servers hosting a TSDS and an EventSpace; the Image Processing application; and both legacy and Java Programmable Logic Controllers (PLCs).

5 Scenarios

To illustrate the TSpaces system in a controls environment, we present several example scenarios for the manufacturing center of The Jelly Belly™ Portraits Company. Jelly Belly Portraits (an imaginary company) manufactures "jelly belly mosaics" using photos submitted by customers. The customer submits an order for a mosaic via the web, along with a digital photo, or a scanned image. Jelly Belly Portraits then builds a custom mosaic with the same image, using one jelly belly per pixel.

5.1 Scenario 1: Asynchronous Device Control and Coordination Through TSpaces

In the first example, we illustrate how TSpaces can be used as a network communication glue for activating the control system to fill an order, as shown in Figure 2. The business process model consists of the company web server, where orders are posted, an Image Processing Service, which processes the photos and image files to create a pixel map for re-drawing the image using jelly bellies, and assembly lines where the mosaics are drawn with jelly bellies.

Each assembly line drawing system consists of jelly belly dispensers, a mechanical arm, and a conveyor. For each mosaic, the mechanical arm moves across the image, dropping jelly bellies into each square on the grid. The jelly bellies are fed to the arm by dispensers, which drop them one-by-one when commanded by a Programmable Logic Controller (PLC). To illustrate the flexibility of the TSpaces approach, we assume that the PLCs are a combination of legacy PLCs and Java PLCs. The legacy PLCs are controlled from a soft controller on a PC, which acts as a protocol gateway ("proxy") to the middleware; the Java PLCs have real-time Java Virtual Machines on the chip, allowing them to talk directly to the middleware.

The challenge for the mosaic drawing system is to reconstruct the image using jelly bellies as "ink" to fill each pixel appropriately. For each mosaic, the image processing stage produces a pixel map encoding each pixel with (color, x, y) coordinates. Assuming a global timing system that lets the PLCs know the coordinates of the mechanical arm as it is scanning, the key challenge is to communicate to the PLCs at which coordinates of the mosaic to drop bellies of a particular color. In our first example we show how TSpaces can be used to coordinate multiple PLCs and multiple assembly lines.

```
<service name="PLC">
    <manufacturer name="Manufacturer">PLCsRUs</manufacturer>
    <model name="Model">X-10-PLC</model>
    <admingroup name="AdministrationGroup">Manufacturing</admingroup>
    <communication name="Communication">
        <type name="Type">TSpaces</type>
        <host name="Host">9.1.25.191</host>
        <port name ="Port">8099</port>
        <space name="Space">EventSpace</space>
    </communication>
    <command name="DropJelly">
        <arg name="Color" type=java.lang.String.class>Red</arg>
        <arg name="XCoord" type=int>15</arg>
        <arg name="YCoord" type=int>20</arg>
    </command>
</service>
```

Figure 3: Example XML service description for a PLC in the assembly line. The service description is registered with the TSDS, and discovered by the Image Processing System.

The proxies create and manage an instance of a Java object representing each legacy PLC. The Java PLCs manage their respective objects in the address space on the chip. We assume that the factory network provides an IP backbone, over which are connected the various parts of the system — the Java PLCs and the Java soft controller proxies, the Image Processing System, a web server, and TSpaces servers running on a PC/workstation host.

At system startup, two setup phases occur, service registration and event registration. The PLCs are each services, made available to other devices in the system, by registering their service descriptions with the TSDS. Service descriptions are an XML document including a root tag identifying the service type (PLC), and named attributes. The service description may imbed the interface schema (addressing information and command signatures) for commanding the PLC, or simply contain a URL pointer to a manufacturer's interface schema on the web. An example service description in shown in Figure 3.

The PLCs register with a TSpace server that buffers events (called an EventSpace). The registration templates specify a color value as a selection key for filtering events notified to each controller.

At run-time, when a new image is processed and a mosaic pixel map is generated, the Image Processing System contacts the TSDS to discover the interface schemas for the PLCs. From the registered service descriptions, the Image Processing System generates tuples conforming to their command signatures, and learns the address of the EventSpace. As the Image Processing System writes events to the EventSpace, the PLCs receive callbacks based on their registrations. The events are commands which cause the dispensers to drop jellies as the arm scans.

Now, imagine that a new PLC with a new color belly is added to the system. The new PLC simply registers its service description with the TSDS registers with the EventSpace. Once the new PLC is discovered by the Image Processing System, it adjusts its color mapping and creates a new event type — while the system continues to run.

5.2 Scenario 2: Device Control from an Information Appliance Using MoDAL

The second scenario shows how MoDAL document interfaces coupled with the TSpaces system can allow a information appliance to be used as a user interface to an arbitrary device. The user has a PDA connected to the Internet via a wireless communication link. (We currently use a CDPD or Ricochet modem.) The MoDAL client on the device bootstraps the TSpaces system and obtains the address of its local proxy from the TSDS. The MoDAL client connects to the proxy and downloads the list of locally available devices. When the user selects a device in the MoDAL application index screen, the proxy contacts the TSDS and obtains the MoDAL document for it from an Interface Repository (which could be a tuple space, or simply a web server) and passes it to the PDA. The buttons and other user interface objects on the PDA are associated with tuples of a particular schema, so that when the user activates the control panel, the corresponding command tuple is written to a TSpaces for reception by the device. As the user moves around the factory floor he/she can continue this process, downloading MoDAL interfaces to devices as needed.

6 Future Work

The VuSystem [14] and MASH toolkit [15] system architectures combine a high-level scripting layer with objects built in a systems programming language to separate control and data handling of multimedia data types. Though multimedia may seem unrelated to the problem at hand, the clean separation of configuration (at the scripting level) and basic function (implemented in the systems language) is a powerful paradigm that could be extended for use with distributed components. We are currently investigating the possibility of importing such a split-object model into the TSpaces system via the use of a domain-specific "scripting" language for defining event exchanges that occur between objects written in Java.

7 Summary

We have described the TSpaces system and its usage for device control in heterogeneous networks of heterogeneous devices. The key principles used in developing the architecture are: 1) the use of gatewaying and IP to support internetworking, 2) event exchange as a communication metaphor, 3) leveraging intermediary proxies to decouple endpoints, 4) using a rich middleware layer and common schema to simplify composition and promote flexibility. The TSpaces system (written in Java) and MoDAL document application language software (for PalmOS) are available for free download from `http://www.alphaworks.ibm.com/`.

References

[1] ASHRAE. BACnet Specification. Atlanta, GA.

[2] J. Bates. A Framework to Support Large-Scale Active Applications. *Proceedings of the ACM SIGOPS European Workshop*, September 1996.

[3] J. Bates, J. Bacon, K. Moody, and M. Spiteri. Using Events for the Scalable Federation of Heterogeneous Components . *Proceedings of the ACM SIGOPS European Workshop*, September 1998.

[4] David Beech et al. XML Schema. `http://www.w3.org/TR/xmlschema-1`.

[5] Bruce Khavar. Java Specification Request JSR-000007 Factory Floor Automation.

[6] Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4), April 1989.

[7] Steven Czerwinski, Ben Zhao, Todd Hodes, Anthony Joseph, and Randy Katz. An Architecture for a Secure Service Dis covery Service. In *Proceedings of the Fifth Annual International Conference on Mobile Computing and Networking*, Seattle, WA, August 1999. ACM.

[8] K. Eustice, T. Lehman, A. Morales, M. C. Muson, S. Edlund, and M. Guillen. A Universal Information Appliance. *IBM Systems Journal*, pages 454–474, October 1999.

[9] D. Gelernter. Generative Communication in Linda. *TOPLAS*, 7(1), January 1985.

[10] Hewlett-Packard. e-Speak White Paper. `http://www.hp.com/go/e-speak/`, 1997.

[11] Todd Hodes and Randy Katz. A Document-based Framework for Internet Application Control. *2nd USENIX Symposium on Internet Technologies and Systems*, October 1999.

[12] Bruce Khavar. Personal communication, September 1999.

[13] Andrew Layman et al. XML-Data. `http://w3c.org/XML/TR/1998/NOTE-XML-Data`.

[14] C. J. Linblad and D. L. Tennenhouse. The VuSystem: A Programming System for Compute-intensive Multimedia. *IEEE JSAC*, 14(7), September 1996.

[15] Steven McCanne et al. Toward a Common Infrastructure for Multimedia-Networking Middleware. *Proc. 7th Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV '97)*, May 1997.

[16] Kelvin Nilsen. Adding Real-time Capabilities to java. *Communications of the ACM*, 41(6):49–56, June 1998.

[17] Ninja. The Ninja Project. `http://ninja.cs.berkeley.edu`.

[18] Object Management Group. Common Object Request Broker Architecture. `http://www.omg.org/`.

[19] Pradip Madan. Overview of Control Networking Technology. Echelon Corporation.

[20] R. Blomseth and W. Capolongo and B. Dolin and J. Lund. LonWorks Network Services (LNS) Architecture Technical Overview.

[21] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. *Service Location Protocol Internet Draft #17, draft-ieft-svrloc-protocol-17.txt*. IETF, 1997.

[22] Jim Waldo. The Jini Architecture for Network-centric Computing. *Communications of the ACM*, pages 76–82, July 1999.

[23] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, August 1998.