ELSEVIER

Computer Networks 35 (2001) 457–472

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Hitting the distributed computing sweet spot with TSpaces

Tobin J. Lehman [a,*], Alex Cozzi [a], Yuhong Xiong [a,1], Jonathan Gottschalk [a], Venu Vasudevan [b], Sean Landis [b,2], Pace Davis [c], Bruce Khavar [d], Paul Bowman [d]

[a] *IBM Research Division, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA*
[b] *Motorola Labs, 1301 E.Algonquin Road, IL02-2240, Schaumburg, IL 60196, USA*
[c] *Brokerage Systems and Services, 68 Lombard Street, London, UK EC3V 9LJ*
[d] *Cyberonix, 777 Oakport Street, Suite 810, Oakland, CA 94621, USA*

Abstract

Our world is becoming increasingly heterogeneous, decentralized and distributed, but the software that is supposed to work in this world, usually, is not. TSpaces is a communication package whose purpose is to alleviate the problems of hooking together disparate distributed systems. TSpaces is a global communication middleware component that incorporates database features, such as transactions, persistent data, flexible queries and XML support. TSpaces is an excellent tool for building distributed applications, since it provides an asynchronous and anonymous link between multiple clients or services. The communication link provided by TSpaces gives application builders the advantage of ignoring some of the harder aspects of multi-client synchronization, such as tracking names (and addresses) of all active clients, communication line status, and conversation status. For many different types of applications, the loose synchronization provided by TSpaces works extremely well. This paper relates our experiences in building distributed systems with TSpaces as the central communication component. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Distributed computing; Java; Database; Messaging; Message oriented middleware; Connectionware; Tuplespaces

1. Introduction – disparate computing

The business world is the center of "disparate computing". Through acquisitions, mergers, decentralized IT shops and just plain miscommunication, most large businesses have numerous incompatible software subsystems. In addition, it is usually impossible (or at least prohibitively expensive) for them to start over from scratch, merge the existing components into one monolithic system or rebuild all of the software components so that they become matched parts of a distributed system. The only real choice a business has is to run its various programs (business planning, accounting, payroll, inventory, etc.) independently and then build ad hoc gateways between them so that they can communicate. Though difficult and messy, the ad hoc gateways do not pose a terrible problem if there are only a few. However, that can change if there are entirely new computing platforms that must be added each potentially with many new programs and devices. For example, consider a business that wants to combine its business software components, such as

[*] Corresponding author. Tel.: +1-408-927-1781.
E-mail address: toby@almaden.ibm.com (T.J. Lehman).
[1] Present address: 4401 Calypso Terrace Fremont, CA 94555.
[2] Present address: P.O. Box 8000, Park City, Utah 84060.

1389-1286/01/$ - see front matter © 2001 Elsevier Science B.V. All rights reserved.
PII: S1389-1286(00)00178-X

payroll, accounting and inventory tracking, with other software and devices, such as the warehouse robot device control system, the building HVAC controls, the building security system, the employees' pagers and smart cell phones. Designed as one large entity, such a large system would present a formidable challenge to build. However, designed as a set of independent components with well-defined inter-faces that are joined with an intelligent messaging middleware system, it could be relatively straightforward to build and maintain.

From a technology *user's* viewpoint, the world is becoming increasingly convenient, with more connected devices in the home and office, more inter-connected and functional programs on the Internet, and more services on the Web. From a technology *provider's* viewpoint, however, the world is becoming increasingly distributed and heterogeneous, which equates to a greater difficulty in building applications that span multiple platforms and connect large numbers of Web or Internet services. One technique for addressing this problem, which has shown considerable promise, is the services paradigm – building all software pieces as independent entities (called services) that communicate via messaging middleware. By keeping the components separate, they can be tested, maintained and improved independently. In addition, by using messages via a middleware component rather than using a direct remote procedure call (RPC), there is an opportunity for adding translators to the communication stream that fix protocol, data format and data schema disparities.

The benefits of asynchronous messaging are well known in the information technology (IT) shops of the Fortune 500 companies. In fact, a recent IDC market prediction suggested that "Message-ware" alone was going to grow 81% over the next 5 years to a 5 billion-dollar market. Why does messaging middleware have such rosy predictions? There are many answers, but the main one is simple – it will allow businesses to be more profitable with the software that they already have, as well as give them the flexibility to grow into new areas without having to completely rewrite their current software systems.

1.1. Hitting the sweet spot

TSpaces [8,9], a project at the IBM Almaden Research Center, is a messaging middleware component that combines asynchronous messaging with database features, all in one easy to use, portable package. Having a very small footprint and being written in Java, it has the ability to run on virtually any platform, from very small devices (like a palm device or an intelligent network hub) to mainframes (like the IBM 390 platform). The database capability of TSpaces sets it apart from other pure messaging systems. By being a persistent data store, as well as a data delivery system, TSpaces provides applications with greater freedom with respect to data management – the message queues are also database stores that can be queried with a sophisticated query language. This gives users the flexibility to use a single system for communication and basic storage needs, and it also gives them the ability to treat transient data and permanent data in the same way.

TSpaces offers users a different style of multipoint communication. Different from multi-cast, where the receivers must register in advance, TSpaces provides something more like *multi-receive*, where receivers can attach "after the fact" and still receive messages. This opens the door for solutions that involve dynamic sets of clients, producers/consumers, agents or service providers.

From monitoring the hundreds of organizations that are using TSpaces, we have seen that TSpaces offers a useful set of features that benefits a wide variety of distributed applications – basically, it hits the "sweet spot" of distributed computing. In this paper, we will present an overview of TSpaces and a simple taxonomy of the types of distributed application that have profited from using TSpaces. They share one or more of the following uses of TSpaces: as a message/event delivery system, load balancing/sharing mechanism, a local system hub, a legacy application integrator, a semi-structured database system, a transcoding host platform, or an application server. Then, we will show how we have used TSpaces in our own lab in several projects. Though we have numerous internal projects: a universal print solution, a message delivery system, a services registry, a distributed file system, an application distribution system, a multi-client PDA synchronization database system, a core for an intelligent environment and a service interface database system, we will highlight three representative projects. Finally, we will explore several case studies of TSpaces in customer products or in production use.

This paper is organized as follows. In Section 2, we will give a brief overview of the TSpaces function and then provide a simple taxonomy of TSpaces usage. In Section 3, we will present examples of IBM solutions that use TSpaces, namely, a universal print solution, an application launcher and a multi-user personal device Sync Server. In Section 4, we will describe several customer applications where TSpaces plays a central role. Finally, in Section 5, we will draw some conclusions from our study of TSpaces applications and conclude the paper.

2. TSpaces function and usage taxonomy

One of the challenges in explaining what TSpaces is, is coming up with a set of terms or categories that define what it does. Since we could not find an existing term or category that accurately describes the TSpaces function or abilities, we coined the term "Intelligent Connection-ware" as a broad category to cover the TSpaces functions of asynchronous messaging, event notification, XML document management and transactional database management. As for coming up with a new term to define TSpaces itself, we have been less successful. While the term "Software Duct Tape" is somewhat accurate (it is a product used to connect things that were not originally made to be connected, or a product used to patch an existing system), it does not quite capture the essence of the flexibility of TSpaces in the general area of distributed computing. We are still looking.

2.1. TSpaces function

TSpaces, a direct decendant of the Linda system [1,2] at Yale University, has a relatively simple programming interface. The primary data structure is the **tuple**, which is a vector of fields, where each field contains a typed value. Tuples live in spaces, which are simply collections of tuples. A user invokes TSpaces operations on a space, reading and writing tuples. Users create spaces, giving them their names and defining their purposes. For example, one space might be used to hold print jobs, while another might be used to hold temperature sensor output, while still another might be used to hold banking transactions.

2.1.1. Tuples – the basic data structure

Each field in a tuple has a value, which can be a primitive type (e.g., integer, string, float) or a complex type (e.g., a multi-dimensional array, a Java object, a Java class). Tuples may even contain tuples (i.e., they can be nested), which provides for greater user freedom in managing data and also provides for greater freedom for the TSpaces server to manage and manipulate data. Everything in TSpaces is managed as a tuple – requests going from a client to the TSpaces engine are wrapped in a tuple, and the results coming back are expressed in a tuple (a tuple of tuples, if the result contains multiple items). In our standard notation, a tuple with three fields (the first field, an integer with the value "5", the second field, a string with the value "bar", and a third field, a float with the value 9.3) would be represented as ⟨**5**, "**bar**", **9.3**⟩.

A TSpaces client writes tuples to the server and also reads tuples. In order to read, the client must specify the tuple(s) to be read with a query. The simplest form of query is a template match. A template tuple comprises both actual fields (fields with values) or formal fields (fields with just a type). For example, to match the tuple ⟨**5**, "**bar**", **9.3**⟩, we could perform an exact match and specify template tuple with the following values: queryTemplate = ⟨**5**, "**bar**", **9.3**⟩, which would match only tuples of the form ⟨**5**, "**bar**", **9.3**⟩. Or, we could do a more general match and specify queryTemplate = ⟨Integer, String, Float⟩, which would match any three field tuple comprising an integer, a string and a float. We could even do something semi-specific, like queryTemplate = ⟨5, String, 9.3⟩.

2.1.2. TSpaces operations

The basic primitive operations supported by the TSpaces server are:

- **write(tuple)**: Adds a tuple to a TSpaces space.
- **take(templateTuple)**: Performs an associative search for a tuple that matches the template. When found, the tuple is removed from the space and returned. If none is found, null is returned.
- **waitToTake(templateTuple)**: Performs an associative search for a tuple that matches the template. Blocks until a match is found. Removes and returns the matched tuple from the space.
- **read(templateTuple)**: Like "take" above, except that the tuple is not removed from the tuple space.
- **waitToRead(templateTuple)**: Like "waitToTake" above, except that the tuple is not removed from the tuple space.
- **scan(templateTuple)**: Like "read" above, except returns the entire set of tuples that match.
- **eventRegister(command, template tuple, callback routine)**: Register for an event corresponding to the command (Write, Delete/Update) and the template tuple. The supplied (client-side) callback routine will be called when the event occurs.
- **countN(templateTuple)**: Like "scan" above, except that it returns a count of matching tuples rather than the set of tuples itself.

There are other operations (e.g., eventDeRegister, getVersion, status, exists, delete, deleteAll, multiWrite, update, cleanup), other functions (admin/security commands, XML support, transaction features, applet support), as well as a mechanism to create new operators and download them, but we will not go into detail on those features here. We refer the interested reader to the Programmer's Manual on the TSpaces Web page [7].

2.1.3. Using TSpaces

To use TSpaces, a client must locate a TSpaces server (either by knowing a server name or by using a server location service) and then attach to a particular tuplespace inside of it. In a TSpaces server there can be millions of tuplespaces (also called "spaces"), which are collections of tuples that are related through the applications that manage them. Unlike a relational database, where tables each have a particular design, or schema, a space does not have any schema associated with it. The tuples in a space are all self-describing, which means they do not rely on any sort of "space schema" to give their values meaning.

For a given use, like communicating with a known device such as a printer, the name of the tuplespace may also be supplied by the TSpaces discovery service. Once attached to a tuplespace, a client calls the various TSpaces operators to manipulate tuples (read, write, take, update, delete, etc.). A client can communicate with an arbitrary number of clients by interacting with them through a single space, but a client is also not restricted to a single space or even a single server. Clients have the freedom to attach to servers and interact with spaces at will – there is no "message channel" set up required and there is no penalty for detaching from a server and reattaching later.

2.1.4. The usefulness of TSpaces

TSpaces embodies a useful set of functions in a single system. It has database capabilities, but it does not require the user to learn about the complexities of the relational model or the difficulty in using the dynamic programming interface. It has messaging capabilities, but it does not require the user to set up complicated message channels. It employs a Web server internally that allows users to view the contents of a space – without having to form a single query. Our own experience with writing TSpaces applications has shown that this simple feature – being able to look inside the TSpaces engine with a Web browser – has reduced debugging time by a factor of three.

So, TSpaces is a useful "Swiss Army Knife" for programming. However, it is not just the things that it does, but also the way that it does them. By promoting asynchonous messaging as its primary method for interaction, TSpaces encourages a highly effective design methodology in application construction. Components connected through TSpaces are loosely connected though a messaging interface. This makes it easy to test components separately just by sending and receiving tuples. Also, since the messages themselves can easily be examined, the communication pipeline can also be checked and monitored, in addition to the individual components.

Asynchronous messaging also promotes disconnected use. Wireless clients, mobile devices, and non-reliable connections work well with TSpaces. A client can connect just long enough to post some messages and receive some messages, without incurring any penalty for disconnecting. Yet another benefit of asynchronous messaging is the notion of anonymous clients that can connect and participate in a group job (e.g., a large parallel computation), or provide some sort of service. By tying the activation of a service to a space, it is possible to trigger the launching of a service from any electronic platform (e.g., a workstation, a phone, a PDA or even another service).

2.2. A simple usage taxonomy

From our own use of TSpaces and from observing the TSpaces customer base, we have identified at least eight distinct basic uses of this technology. Most applications incorporate multiple aspects (such as a message system, event notification system, XML store and database system). We list the eight main uses of TSpaces here, each with an explanation.

- *Heterogeneous message system.* Applications needing to send messages across multiple platforms (e.g., Linux, MacOS, VM, AIX and even Windows) use TSpaces to send and receive information. An example of this is IBM's InfoPrint Manager product, which uses TSpaces to send print job status messages between its Windows and Unix components.
- *Event mechanism for heterogeneous platforms and disparate devices.* Applications needing to be notified on one platform or device when something happens on another platform or device can use TSpaces as the "event intermediary".
- *Load sharing or load balancing.* In any situation where there are producers and consumers, or multiple agents to perform a stream of tasks, TSpaces is an ideal mechanism for sharing jobs across multiple workers. One or more work creators submit jobs by placing "job tuples" in the job space, and available workers remove them and execute them as needed.
- *Object database, semi-structured database or transaction system.* TSpaces is a tuple-based database system. Although it is neither a relational database system nor an object-oriented database system, it behaves a bit like both. A user can treat a tuplespace as a set of homogeneous records, much like a relational database table. However, the user is not required to predefine the schema of the records, as he would in a relational database table [4–6]. A user can also treat a tuplespace as a collection of objects – a user can store any Java object (or graph of objects) in the field of a tuple, but still perform indexed searches on those objects.
- *Local network controller.* In a multiple heterogeneous appliance closed environment, such as an automobile, home or factory, TSpaces provides the communication layer, the device status broadcast mechanism, the permanent log of message activity, the authorization layer and the device protocol/data format message translator. In embedded applications, where devices know only how to speak their own language and say a limited number of things (e.g., "The coffee is ready", "Sensor 2a-493 reading at 14:00 is 451°F", or "Pod bay door malfunction"), TSpaces performs the job of message clearinghouse and universal translator.
- *Gateway and host for transcoders and transmogrifiers.* [3] As an asynchronous messaging component, TSpaces represents a natural platform for intermediaries, proxies, filters, generators, monitors and transcoders. Messages passing through TSpaces can be modified by multiple translators in order to be more presentable to the receiving party.
- *An XML Store.* Though not intended to serve as an XML warehouse (TSpaces is meant to store megabytes of data, not terabytes or petabytes), TSpaces does have the ability to store XML documents and retrieve them using the XML query language, XQL. Our intended use for the XML feature is to serve as a service

[3] A transcoder is something that changes the shape of an object or information. For example, a transcoder might change the order of the attributes of an address record, or it might change the protocol that is used to send the data. A transmogrifier, on the other hand, performs a complete change of an object (say, from a personnel record to a dung beetle).

description mechanism, where network services are described in XML and potential clients can search the list of services using XQL.

- *A service discovery mechanism*. As described in the above use (An XML Store), TSpaces works as a discovery mechanism. Services (devices, programs or resources that conform to the standard service API) register themselves by entering a description tuple into a predefined space in the TSpaces server (Usually the TSDS – the Tuple-Space Discovery Space). Clients query the discovery space to locate services and employ them.

3. Uses of TSpaces inside of IBM

At the IBM Almaden Research Center, the birthplace of TSpaces, we have created numerous solutions with TSpaces at the core, three of which we will describe here. We have created a universal print solution, an application launcher, and a multi-sync server (MSS).

3.1. A universal print solution

Despite the decades of progress in operating systems, networks and printers, we still had a problem with printing at the IBM Almaden site. Even though we had full connectivity, the simple fact was that we could not easily print on any printer from any client machine. The larger printers were all attached to our AIX (IBM's Unix System) and VM networks, but the smaller printers were attached to individual machines, each running a different operating system (flavors of Linux, Windows, OS/2, or even MAC OS). Any one operating-specific solution, such as running LPD on each machine, was not sufficient.

We encountered several problems:

1. We could not send an arbitrary print stream to any printer – some were postscript printers, some were ink jet printers, some were larger laser printers. We needed a way to convert the print stream created by a client into one that worked with the desired printer, without loading all 300 printer device driver types onto all 1000 client machines in the building. Other solutions involving the dynamic loading of device drivers did not work on our *existing* machines or printers.
2. We needed a mechanism for discovering all available printers and their capabilities. In the days when everyone was using AIX, whatever names popped up when the LPQ command was issued (query all printers) was all the users got in terms of helpful information. Although there was some attempt at helpful naming of the printers (e.g., B1-4029ps meant that the LexMark printer, model 4029, found in the first floor of the B-wing, was a postscript printer), it was just as common for printers to be called "xyzzy" or "TJLps". There was no database of printers kept with useful information supplied about each one.
3. There was no way for a client device that had no print capability (i.e., no capacity to load and run print drivers), such as a Palm device or a smart phone, to contact a printer.
4. Program support for building services, such as printing, was operating system specific. So, if one were to write a multi-platform program that included printer support, then that program would have to perform a different set of tasks for printing, depending on the platform it was on.

What was needed was a mechanism to first find every printer on the network and then send the print jobs to the printers. The solution that presented itself was to use TSpaces as the data connection to clients and printers on all platforms. Since TSpaces is written in Java, it runs on virtually all platforms and has a uniform interface, namely, tuples. As shown in Fig. 1, the major functional pieces in our solution are the Client Daemon, the Printer Daemon, and the file conversion services. Each Printer Daemon controls one or more printers. It registers its set of the printers to TSpaces. By keeping a richer set of information on the printers (and all of the other services, for that matter), it made it much easier for users to figure out where the printer was, who owned it, who maintained it, what the characteristics were (speed, color, duplex, etc.) and just how busy it was. The Client Daemon runs on the user's computer. It detects print requests from the user,

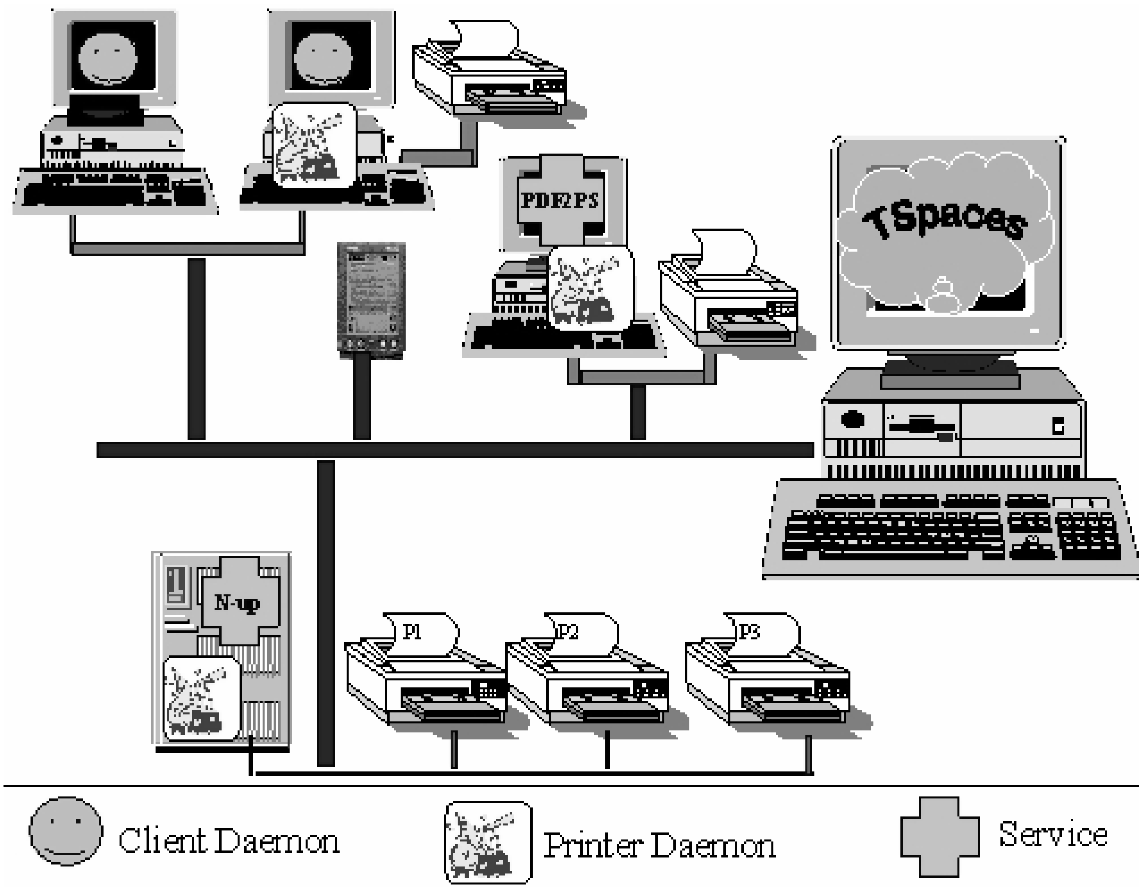

Fig. 1. A universal print solution.

obtains all the available printers and their attributes from the print space, and interacts with the user to choose a printer and its settings. If necessary, file conversion can also be done through the available services. For example, if a PDF file is sent to a PostScript printer, the Printer Daemon can invoke the PDF2PS service before printing.

On desktop computers, the files to be printed are copied to a predefined print directory. The Client Daemon keeps polling for new files in this directory, and, when it sees one, brings up a print wizard on the screen. The wizard shows a list of the printers registered in TSpaces. The user can then choose a printer, and send the print job to the space. On hand-held devices where thread support is not available, the client software simply writes a tuple with the print job to TSpaces. That job will be picked up by another computer for processing. On the printer side, the Printer Daemon obtains the printer information, such as the printer name, print command, printer features, from an XML file. After registering the printers, it blocks waiting for print jobs. When a job arrives through the print space, it first checks if the printer can directly handle the format of the print file. If not, it first performs file format conversion through a conversion service before sending the print command to the printer.

This architecture has several advantages over conventional printing mechanisms on Windows and UNIX systems. On Windows, a user must install the drivers for all the printers he wants to use, which is troublesome. The benefit of this, though, is that the user can get access to printer specific features. For the LPD/LPR protocol on UNIX, driver installation is not required on the user machine, but the user does not get access to printer specific features. TSpaces printing gives the user the best of both worlds. By maintaining a loose coupling between clients and printers, it is not necessary for clients to load printer-specific device drivers. A user can obtain printer information from an XML file passed through the print space. When new printers are added, the Client Daemon will automatically find it. Also, clients have access to printers anywhere in the network, anywhere in the world. Since we regularly work with other IBMers in other research and development labs, it is convenient to send print jobs to coworkers at *their* printer, halfway around the world. Moreover, since TSpaces also connects services, powerful transformations can be achieved in a printing process. For example, a user does not have to have the Adobe Acrobat Reader installed in order to print PDF files. Also, a user can request to print multiple logical pages on a single physical page (the *N*-up format). Traditionally, a user must find a computer where such a conversion program is available, and transfer the file back and forth before printing. With TSpaces printing, the user does not even need to know where the conversion is performed. Furthermore, some advanced features can also be implemented besides file format transformation. For example, a service can filter out all the color printers, or find the printer located closest to the user. All of these make printing much easier and pleasant.

3.2. PIOVRA – an application launcher

Piovra is an application distribution and monitoring system based on TSpaces. Piovra consists of a small client that is distributed in the form of a jar file and is installed on the client machines. The Piovra client is a "launcher" program that connects to a TSpaces server and downloads the list of the available applications. On a user's request the launcher download the classes that compose a Java application and runs it in a separate thread. The applications are stored on the TSpaces server, where each class file is stored in a different tuple (see Fig. 2).

As opposed to the applets distribution model, which is based on a Web server where applications are downloaded from the Web and reside inside a browser, or the Web cast (push) distribution

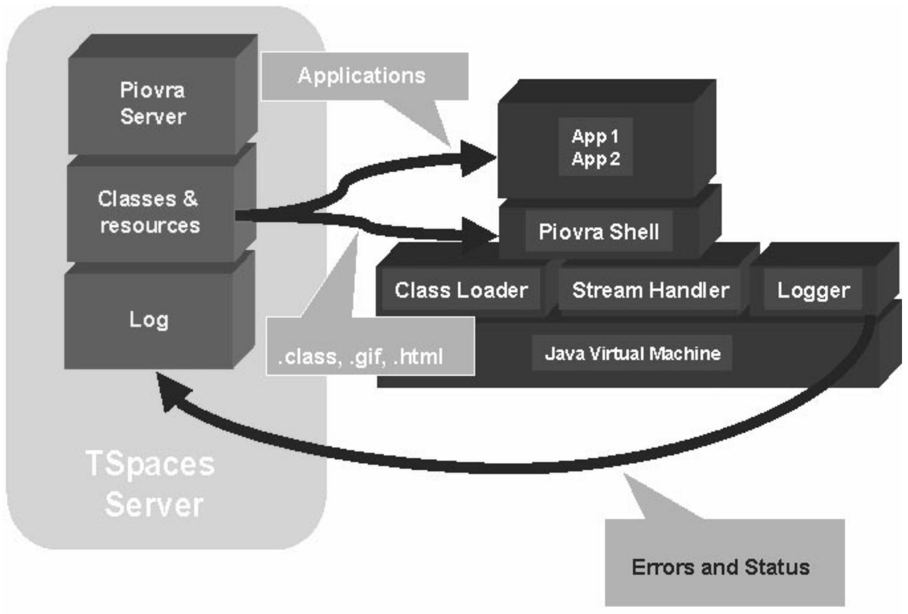

Fig. 2. The Piovra Java application launcher.

model, TSpaces is a bi-directional communication channel. A Piovra client is notified each time a new application or a new release is installed on the server or removed from it, and the Piovra client can report to the TSpaces server various statistics regarding the running applications. In addition, Piovra is not limited to simple applets – it is able to distribute full-fledged Java applications. Any pure Java application can be run using the Piovra environment without changes.

The classes stored into tuples are augmented with meta-information, like the class name, the revision and the grade of maturity of a particular class. This way it is easy to design different clients for different kind of uses. The developers, for example, can use the Piovra client in "developer" mode, having access to all of the latest releases of each class, no matter how unstable they are. The normal users on the other hand will see a new release of an application only after all of its classes have been marked as "stable".

Since the launched Java applications share the same Java Virtual Machine of the launcher, the memory footprint and the startup time of Java applications is dramatically reduced. This allows more applications to run at the same time, using a fraction of the resources normally required when the same applications run in separate JVMs.

An additional service provided by the Piovra environment is that the launcher catches very exception and every log generated by the launched applications and reports it back to the TSpaces server. This way precious information about the reliability and the performances of the deployed applications can be easily collected.

All these features make Piovra an excellent way to distribute and manage Java applications inside an intranet, giving to system administrators new levels of flexibility and control: all the applications are kept in a centralized location on the TSpaces server, thus enabling the rapid deployment and upgrade of new applications.

3.3. Multi-sync server

Today's work force enjoys the benefits of mobile personal information managers (PIMs) such as Palm Pilots, cell phones, pagers, and laptops. Yet the convenience of these devices comes at a cost. The average PIM user stores information on these devices as well as more robust desktop-based PIMs. The use of multiple PIMs then creates multiple, independent storage locations for information. This problem introduces a doubt and certain nagging questions to the PIM user: "where did I put my broker's number" or "did I update this week's meeting on my palm AND outlook?"

There is an obvious need for synchronization. Synchronizing data from multiple PIMs simplifies the user's interaction with these devices and removes that nagging doubt. Without synchronization, the PIM user must perform a manual copy or edit for every bit of information added to a PIM. Via synchronization, the PIM user can be guaranteed that any bit of data entered in any device will be available and current on any other PIM.

The synchronization of PIMs for an individual PIM user can be extended to the synchronization for multiple PIM users each with multiple PIMs. A group of PIM users shares the same set of problems with the single PIM user: information entered in a single PIM lacks the means to reach other PIMs. Yet the group has the added problem of information lacking the means to reach the PIMs of other group members. Again, synchronization comes to the rescue but, in the group setting, it must address some additional concerns. First, since multiple users are involved, the security of personal information must be maintained. Also, an individual PIM user in this group must have the option to reject unwanted group information before it reaches the memory-scarce landscapes of the PIM.

The TSpaces MSS accomplishes individual and group synchronization simultaneously while addressing the concerns cited above. In its initial release, the MSS synchronized on these PIMs: Palm Pilot/Palm Desktop, Lotus Notes, Microsoft Outlook, and a Web Based PIM, anyday.com. The synchronized information is of the standard ubiquitous PIM data types: "to do", "memo", "address/contact", and "calendar/events" (see Fig. 3).

In the implementation of the MSS, there exists a module for each PIM that collects information (to be passed to the server) from the PIM and stores information (received from the server) in the PIM. The *client manager* and accompanying *GUI*

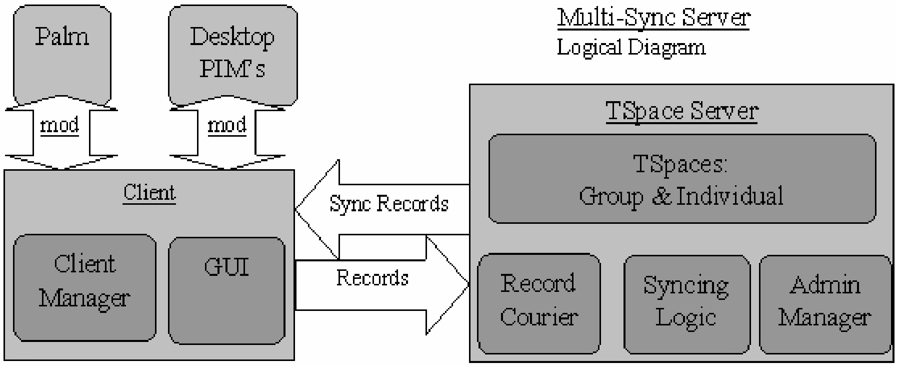

Fig. 3. The IBM TSpaces multi-sync server.

will call these modules in addition to forwarding administrative requests (create group, change password) to the server. The server's *record courier* delivers individual records into their corresponding tuplespaces from an incoming set. These tuplespaces represent any group to which a PIM user belongs or the PIM user's personal tuplespace. Synchronization is then performed on each of these tuplespaces. The server then aggregates records from each of these tuplespaces and sends them back to the client manager. The client manager again calls the modules to store the newly synchronized information.

4. Uses of TSpaces outside of IBM

Hundreds of organizations have downloaded TSpaces from the Alphaworks URL (http://www.alphaworks.ibm.com/) and tried it out. From their feedback, we have learned that often, to their (pleasant) surprise, it not only works for them instantly "out of the box", but it also solves their distributed programming problem. In this section, we relate the stories of three different customers that tried TSpaces and found that it solved their distributed programming problems.

4.1. Motorola's Mojave system – mobile agents for wide-area systems management

Current approaches to systems management (e.g., SNMP) rely on a benevolent network environment where latency is predictable, failures rare, bandwidth plentiful, and attacks absent. These assumptions are increasingly inaccurate for today's communications applications that may be globally dispersed, may run over a mix of wired and wireless networks, may operate over public and private network resources, and use off-the-shelf hardware and software. Managed applications that operate in dynamically changing conditions impose new requirements on the management platform. The management software has to be malleable. In other words- internally resilient to changes in its environment and capable of reconfiguring itself (internally and dynamically) to effectively manage in a changing environment. The Mojave project aims to build malleable systems managers by using mobile, environment-aware agents as the component technology.

Mobile agents are components that are environment-aware (network, resource and threat aware), have an independent thread of control, and are capable of autonomously moving while carrying their computational state with them. The combination of autonomous mobility and environment-awareness is key to attaining malleability. If one can build environment-awareness into individual components of the systems manager, then they can monitor when their own performance is sub-optimal and move to a more optimal location. In the systems management context, an agent may not be getting timely or consistent access to data, sufficient computing resources, or its operation might be hindered because collaborating agents are not co-located. In each case, an autonomous mobile agent can identify (and move to) a more suitable location autonomously, without consulting a centralized mobility manager. The decentralization of environment-awareness distinguishes mobile agent approaches to reconfigurable systems from other competing approaches. Decentralized adaptation is likely to be more accurate (based on better information) and more efficient (as each component of the application adapts concurrently).

The Mojave implementation uses Jini and TSpaces to support a hub-spoke framework for agent mobility and communication (see Fig. 4). Agents run in agent virtual machines, called pods, which in turn map to physical machines. Pods interface to a single (logical) liaison, which is implemented by a TSpaces space. Agents, pods and the liaison are Jini services that register with the Jini Lookup Service (JLUS). Jini lookup and proxy downloading mechanisms are used to tie pods, liaisons and agents together. The TSpaces hub is used to implement both inter-agent messaging and agent migration. Executing agents migrate between pods by migrating in (and out) of a space that is accessible both to the source and destination pod. The source pod stores a migrating agent as an "agent migration" tuple. The state of the migrating agent is a MarshalledObject field in the migration tuple. (Target) Pods that are willing to host incoming agents use the TSpaces callback mechanism to subscribe to "agent migration" tuples. TSpaces query support is usable to implement conditional agent migration, i.e. whereby an agent migrates to any destination pod that provides a certain kind of environment (e.g., low CPU utilization). Conditional agent migration is important to malleability and hard to implement without the

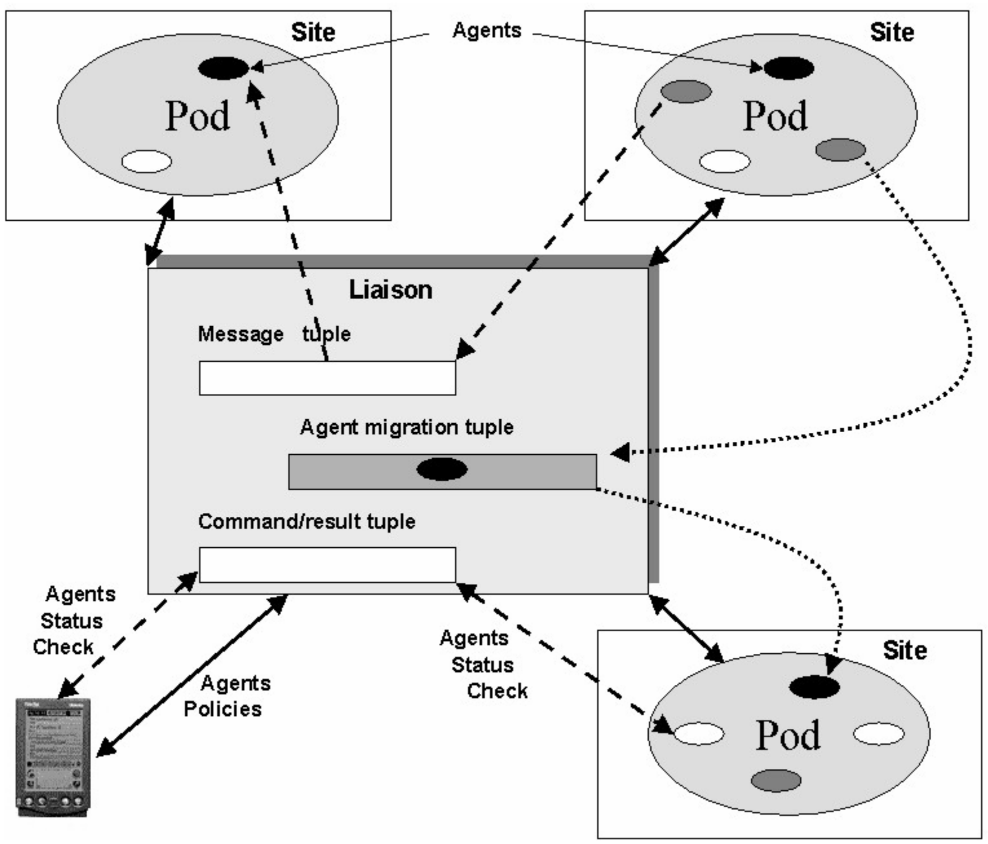

Fig. 4. Motorola's Mojave system.

querying capabilities of a tuple space. Using a tuplespace as the Mojave hub makes it easy to build an agent console that monitors a running agent computation. Since agent migration and messaging are expressed as tuples, a console application can subscribe to all (or selected) events in an agent computation using standard tuple space primitives. The hub-spoke architecture is thin-client friendly, as interfacing to a TSpaces server is a lightweight enough operation to be accomplished from a Palmtop. Mojave demonstrates a KVM (Palm) based monitoring interface that dispatches command tuples and receives result tuples using an extensible Palm HotSync daemon that interfaces to the TSpaces hub of a running Mojave platform.

4.2. Brokerage systems and services – an XML-based equity trading solution

Brokerage systems and services (BS&S) has built a real-time equity trading system using TSpaces as its middleware product. The system is used to allow trading on different stock exchanges from a variety of different front-end interfaces, including HTML and WAP. Having chosen TSpaces as the middleware product from the beginning, they were able to take advantage of the power and simplicity of the tuplespace-programming model immediately. This allowed them to build an extremely fast, flexible system very quickly (see Fig. 5).

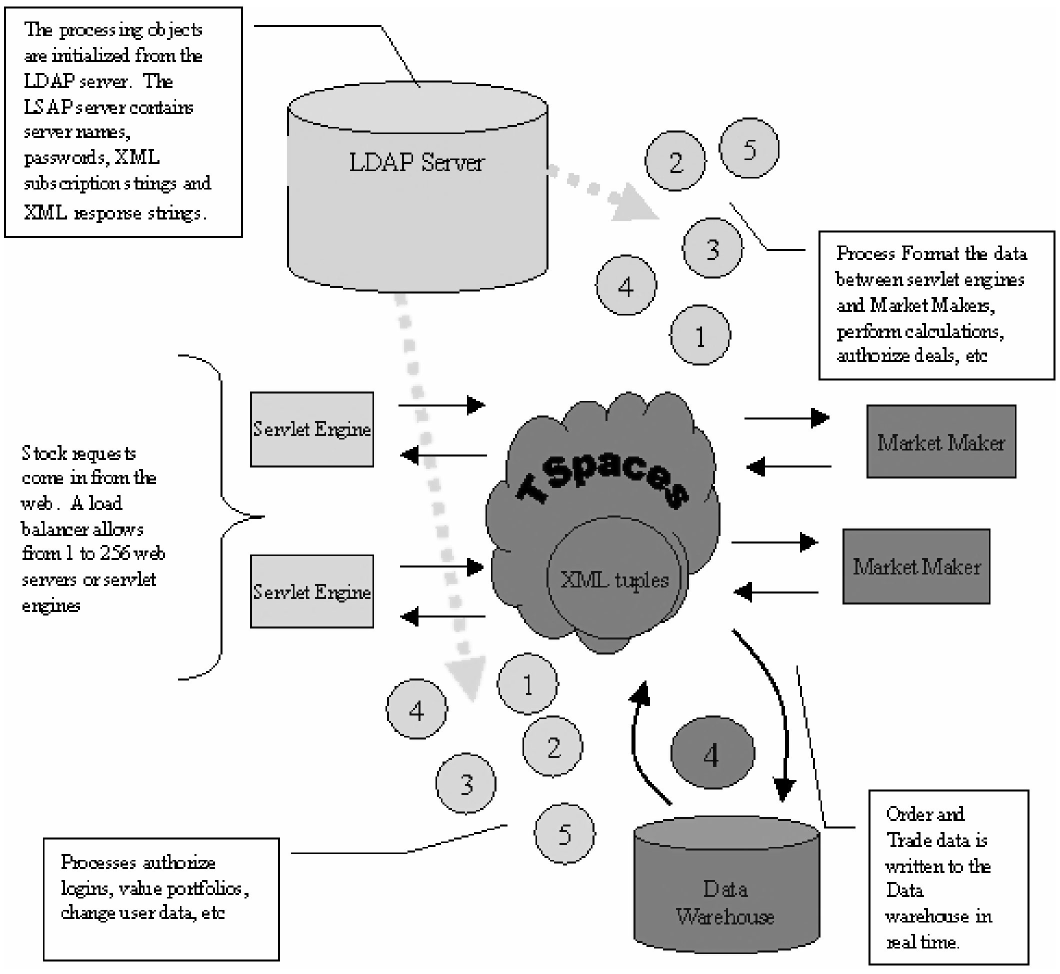
The processing objects are initialized from the LDAP server. The LSAP server contains server names, passwords, XML subscription strings and XML response strings.
LDAP Server
2
5
3
4
1
Process Format the data between servlet engines and Market Makers, perform calculations, authorize deals, etc
Stock requests come in from the web. A load balancer allows from 1 to 256 web servers or servlet engines
Servlet Engine
Servlet Engine
TSpaces
XML tuples
Market Maker
Market Maker
4
1
2
3
5
4
Processes authorize logins, value portfolios, change user data, etc
Data Warehouse
Order and Trade data is written to the Data warehouse in real time.

Fig. 5. Brokerage systems and services.

To begin with, BS&S uses TSpaces XML handling as the main method for passing messages. Java objects are used for information that remains in the spaces on a more permanent basis. Every process is designed to allow from 1 to N instances to be run in parallel. This allows for faster execution and increased fault tolerance by spreading the services across several machines. Apart from the servlet engines, the processes involved can be run on any of the servers within the network. This feature was very simple to implement using the TSpaces model of programming.

Fig. 5 provides a high-level overview of the system. The four main sub-systems are the front-end servlet engines, the back-end market makers, the data warehouse, and the LDAP server. The processes in-between control and format the information as it flows through the system.

As an example of how the system works we will walk through the case of a user requesting a price quote on a stock.

1. The user posts his stock selection to the servlet engine from an HTML form.
2. The servlet engine receives the request and parses the necessary data from the form.
3. The servlet engine then receives a unique identifier for the quote request by taking the QuoteReferenceId tuple out of the space, reading the current id, iterating it by 1, and then putting the tuple back into the space.
4. The servlet then packages up the quote request into an XML message, writes it to the space, and then does a waitToTake on the Response with the supplied reference id.
5. A formatting process then takes this quote request and performs a look up of the market makers that will provide a price for this certain stock. It then formats the request into the proper format for each market maker and writes it to the proper space.
6. The market maker process takes the request and fires it out to the third party.
7. The market maker receives the reply, adds the XML header and writes it into the space.
8. A broker process takes the response, calculates the total price, commission, taxes, etc., and packages it into the proper XML message and writes it into the space with the XML header that the servlet engine has been waiting for.
9. The servlet engine transforms the XML into HTML and writes it to the HTTP response object.
10. The user receives his quote request.

Every process on startup looks for its entry in the LDAP server to initialize itself. The process receives, among other information, the XML query that it will pass to TSpaces to register itself as a listener. It also receives the space name and the XML Header required to append to the beginning of each message written back to TSpaces. On startup every process registers for "Command" events. Command events can include process specific commands as well as a re-init command. The re-init command tells the process to re-read its startup parameters from the LDAP server. This architecture allows for dynamic ordering of the message flow through the system.

The data warehouse is used for analysis and long-term storage. Through out the system each process takes certain information from different messages and fires a copy into a separate space. Different data warehouse processes then take and write into the RDB.

TSpaces was also helpful in the BS&S day-to-day development. The BS&S programming team could maintain several different versions of the system using the same TSpace server. This was achieved by simply changing the naming scheme of the XML headers written into the messages. Team A could be testing the pre-production release while Team B was just rolling out a new proof-of-concept design. This made BS&S more efficient with fewer computing and IS staff resources.

We have described a very simple example of how BS&S uses the TSpaces technology to deploy a robust, efficient trading platform. Although there many more agents and algorithms comprising the complete system, they are not described here, we hope that it demonstrates the power and simplicity of the TSpaces programming model.

4.3. Cyberonix – ECP creates a true mirror world

Cyberonix, a California corporation, is a technology and solution vendor intending its products and services for the global market place. Its core competency is "Embedded Scaled Automation", and it offers products as well as solutions to the global enterprise. The solutions are vertically oriented technologies like Gas Station Automation, factory floor automation, etc.

Cyberonix created Enterprise Common Protocol (ECP), an integration product, to address the needs of an enterprise to obtain real-time awareness of its operation and optimize its operation in an adaptive and swift manner. Based on Java and targeted for the Internet, ECP employs a publish/subscribe paradigm with real-time facilities that can function as a real-time middleware component, as well as a common ground for device integration and interfacing. ECP has application development facilities and tools for a multitude of implementations.

ECP uses TSpaces to provide a flexible, information neutral environment to communicate information between loosely coupled systems and to provide a flexible persistence mechanism that can support ECP messages and objects. The TSpaces persistence and flexible callback system allows ECP control environments to implement object models that utilize work-flow staging or multiple production lines in a manner consistent with the physical flow of work. TSpaces also provides a way to share objects with stateless data paths such as browser pages and bridge with XML object representations.

TSpaces provides ECP with a mechanism for dealing with disconnected clients. In modern control and manufacturing environments many devices that are involved in the control process do not maintain a continuous connection to ECP information channels. Examples of the devices are Automatic Guided Vehicles (AGV's), PDA's, portable batch terminals, barcode readers and transponders. TSpaces allows ECP to provide a communication pathway that maintains active connections between cooperating parties.

The ECP paradigm is designed to help the "global enterprise" to move toward a global event driven operation. The spectrum is from ECP enabled embedded systems, factory floor devices all

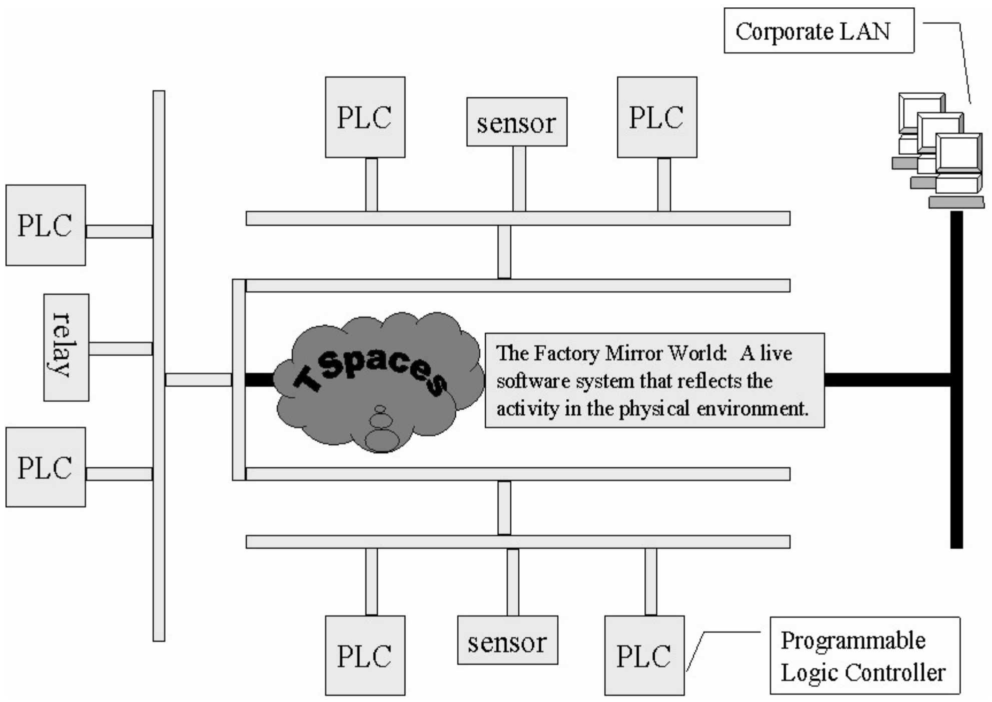
Corporate LAN
PLC
sensor
PLC
PLC
relay
PLC
TSpaces
The Factory Mirror World: A live software system that reflects the activity in the physical environment.
PLC
sensor
PLC
Programmable Logic Controller

Fig. 6. Cyberonix and ECP.

the way to the middle layer and upper IT environment. ECP will also provide facilities toward real-time Internet integration of embedded operations.

In essence, Cyberonix Corporation has used TSpaces to create a true implementation of a "Mirror World", as described in David Gelernter's book, Mirror Worlds [3]. A Mirror World is a software system that represents a virtual copy of a physical system, such as a town, a factory, or a store. The benefit of a Mirror World is that a person really can be everywhere at once, because it is possible to monitor multiple virtual locations or systems anytime from any vantage point.

Fig. 6 shows an instance of a Factory Mirror World, where Programmable Logic Controllers (PLCs) are hooked to various LANs, but also linked to a TSpaces server. The state of the PLCs, sensors and other devices are reflected in the spaces in the server, so anyone on the corporate LAN can query the server to monitor the state of any part of the factory. In addition, since the PLCs, actuators and relays can be directed from TSpaces as well, a remote person could not only monitor the factory but also control all aspects of it as well.

5. Conclusion

We have given a brief overview of TSpaces and provided a simple taxonomy of the primitive functions (the basic uses) found in TSpaces applications. We have looked at case studies of six systems – three from inside IBM and three from outside IBM. Included in the six systems mentioned are the 8 basic uses of TSpaces: a heterogeneous message system, an event mechanism for heterogeneous platforms and disparate devices, a system for load sharing or load balancing, object database, a semi-structured database or transaction system, a local network controller, a gateway and host for transcoders and transmogrifiers, an XML Store and a Service Discovery Mechanism.

For such a simple system, TSpaces has an amazing number of uses in distributed application programming. In fact, as we have learned more about distributed applications, both large and small scale, it's been rare to find an application that would not benefit from TSpaces playing a role somewhere. The fact is, programmers appreciate simplicity. If they can get by with one system instead of two, then they pick the one. TSpaces provides so many essential features (messages, events, database, XML, etc.) in one package, that often solves most of the programmer's distributed application problems. Because of that, we claim that it does, in fact, hit the sweet spot of distributed computing.

In our own research, we are building several projects on top of TSpaces in order to make some major strides in improving Internet computing. With a communication network like TSpaces to augment the Web, we are actively working towards a mechanism to glue more software components together dynamically, which would give us all significantly increased function at a reduced cost compared to today's static systems.

References

[1] Carriero and Gelernter, Linda in Context, Comm. ACM 32 (4) (1989).
[2] D. Gelernter, Generative communication in Linda, TOPLAS 7 (1) (1985).
[3] D. Gelernter, Mirror Worlds, Oxford Press, Oxford, 1991.
[4] The main URL for IBM is http://www.ibm.com, but the DB2 specific material can be found at http://www.software.ibm.com/data/db2/.
[5] http://www.informix.com.
[6] http://www.oracle.com.
[7] The TSpaces Programmer's Guide, located at http://www.almaden.ibm.com/cs/TSpaces/html/ProgrGuide.html.
[8] P. Wyckoff, S. McLaughry, T. Lehman, D. Ford, TSpaces, IBM Syst. J. 37 (3) (1998) 454–474.
[9] http://www.almaden.ibm.com/cs/TSpaces.

Tobin J. Lehman (Toby) joined the IBM Almaden Research Center in 1986, shortly after finishing his Ph.D. degree from the University of Wisconsin-Madison. Toby's research interests include server-based backup systems, Object-Relation database systems, large object management, memory-resident database systems, Tuplespace systems, and just about any cool thing written in Java. Toby is the leader of the TSpaces project and is currently organizing a world domination project based on TSpaces.

Alex Cozzi received a Ph.D. degree in computer science from the University of Dortmund (Germany) in 1998, and a M.S. degree in computer science from the Università degli Studi, Milano, Italy, in 1994. He performed the thesis work at IRST (Istituto per la Ricerca Scientifica e Tecnologica), Italy. His research interests include computer vision, robotics and object oriented languages.

Yuhong Xiong received the B.S. degree in electrical engineering from Tsinghua University in China in 1990 and the M.S. degree in electrical engineering from the University of Washington in 1993. He is currently a Ph.D. student in electrical engineering at the University of California, Berkeley. From 1993 to 1996, he worked at Acuson Corp. as a software engineer. He has also worked as an intern at Synopsys and IBM. His research interests include component-based design, type systems, and distributed computing.

Venu Vasudevan is a Member of Technical Staff at Motorola Labs, where he leads the Mojave mobile agents project, and is involved in projects related to IP-based communication architectures and home networking. He is interested in issues related to the construction of large scale, pervasive distributed computing platforms. He received a Ph.D. from Ohio State University in 1990.

Sean Landis is a senior staff software engineer at Motorola Labs, where he researches Mobile Agents and their application to service provisioning, systems management, home networking, and personal mobile services. He received a Masters of Engineering, Computer Science from Cornell University, and a Bachelors of Science, Computer Science from University of Utah.

Pace Davis is the head of development at Brokerage Systems & Services, which is a UK based software company that designs, builds, and hosts trading systems for the retail brokerage industry.