TSpaces Services Suite: Automating the Development and Management of Web Services

Marcus Fontoura, Toby Lehman,
Dwayne Nelson, Thomas Truong
IBM Almaden Research Center
650 Harry Road San Jose, CA, 95120, USA
+1 (408) 927-1416
fontoura@almaden.ibm.com

Yuhong Xiong
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304
yuhong.xiong@hp.com

ABSTRACT
Web services allow authorized entities (including individuals, corporations, and automated agents) to employ software components created by other parties scattered across the globe. In support of this powerful model of interaction, we have designed and implemented an infrastructure and a set of tools to simplify the development and management of Web services. This infrastructure and these tools form the TSpaces Services Suite (TSSuite). This paper shows how TSSuite supports the Web services model, and it describes the design and functionality of the main TSSuite components.

Categories and Subject Descriptors
D.2.3 [**Software Engineering**]: Design Tools and Techniques – *modules and interfaces, object-oriented design methods.*

D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments, interactive environments.*

General Terms
Design, Standardization, Languages.

Keywords
Web services, Tuplespaces, Service Development, Service Maintenance, Development Environments.

1. INTRODUCTION
The evolution of distributed, Internet-based applications has reached another milestone: Web services. Web services represent a new level of inter-program (and inter-component) interaction and behavior, as they provide a way to build large, flexible applications dynamically, from a set of standard independent software parts. Recently several standards and protocols for Web services have been proposed [8], [9], [13], [14], [19], [21], [22], [23] and Web services have gained a lot of attention by both researchers and practitioners. However, there is still the need for infrastructures and tools that support the Web services paradigm. Infrastructures for the development and management of Web services should be based on widely accepted standards and interfaces, so that the services from different vendors can inter-operate. Tools should assist service developers and providers in automating administrative tasks, such as service registration.

This paper presents the TSpaces Services Suite (TSSuite), which

is an infrastructure and a set of tools for the development and management of Web services. From the infrastructure point of view, TSSuite can be viewed as a new layer over TSpaces [24] that extends the tuplespace model [6] to handle Web services as first class citizens. From the tools viewpoint, TSSuite automates the development and management of Web services based on Java. TSSuite is based on accepted industry standards for Web services, such as UDDI [14], WSDL [23], and SOAP [21].

The rest of this paper is organized as follows. Section 2 presents an overview of Web services and the technologies used to implement them. Section 3 describes some related efforts in the area of middleware and tool support for Web services. Section 4 discusses technical details, including a TSpaces overview and the design and functionality of the main TSSuite components. Section 5 shows the TSSuite in action through a small example: a file type conversion service that converts PDF file to Postscript (PS). Finally, Section 6 concludes the paper and points out our future research directions.

2. BACKGROUND AND TERCHNOLOGY OVERVIEW
2.1 Background and terminology
The core entities related to Web services are services, service interfaces, service brokers, and clients [11], as shown in Figure 1. A **service** is a software application or hardware device that can be invoked or used across a network. Each service implements one or more **service interfaces**, which are described through **service description languages**. A service interface describes the set of **operations** supported by the service. A service **broker** is a registry where services can be published and be discovered by the clients. In fact, a broker is just a special kind of service that provides operations for registering and discovering services. A **client** is a software application that uses one or more services. In general, a client uses a service broker to discover the appropriate information about the service (e.g. its interface, address that it can be invoked) before it can invoke the service directly or through an intermediary. A client can itself be a service. In this case it is a **composite service,** since it uses one or more services as its components.

From the process perspective, the core activities related to the development and management of Web services are creation, deployment, configuration, and monitoring. **Service creation** involves the implementation of the service software and the specification of its interface in a service description language (no order is implied here, and these activities are usually done concurrently). During implementation, the activity of creating a client by integrating services together is called **service**

composition. **Service deployment** includes the task of making the service available in the Internet, i.e., binding the service to a physical address where it can be invoked, and the task of registering the service in one or more brokers (**service registration**). **Service configuration** is an optional activity in which service the configuration parameters are specified (e.g. the default language in a text processor service). Finally, **service monitoring** is the activity of checking the service availability and performance.
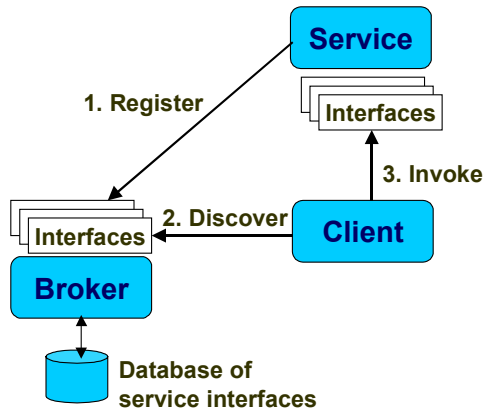

Figure 1. Components in Web services

Service discovery and invocation are activities that happen at runtime. **Service discovery** is the interaction between the client and the broker to discover services and their information. **Service invocation** is the interaction between the client and the service for invoking the service operations, once the client has all the necessary information about the service.

To illustrate these concepts, let us suppose that a librarian wants to buy a large volume of books. Without the support of Web services, she would have to go through the time consuming task of searching several online bookstores for good deals before deciding where to buy each book. On the other hand, if several bookstores are deployed as services, and if these services are published in one or more service brokers, the librarian's task would be greatly simplified. She would develop a client (or use one previously developed) that would query service brokers for discovering all services that implement a given "selling book" interface. This service interface would provide operations for finding book prices and reviews, manipulating shopping carts, and performing the actual buying transaction. During service discovery, the brokers would return a list of all bookstore services to the client. The client could then invoke the appropriate service operations in each of these services to select the ones that provide the better deals, and to actually perform the buying transactions.

All interactions among the clients, brokers, and services are done programmatically, i.e., without human interference. The different systems (service, service broker, and client) are connected through some form of message passing. This kind of interaction is called system-to-system integration, which is one of the key technologies supporting business-to-business (B2B) e-commerce. It allows buyers (clients) to search for and interact with a large number of sellers (services), which would be impossible without such integration.

2.2 Standards and technologies

Numerous standards and technologies are cooperating and competing in the Web services arena. For example:

- The services can be implemented directly in a programming language, such as Java, C++, or C# [19], or as components in a component-based framework, such as COM [20], CORBA [17], or Java Beans [4];

- The service interface can be described using the Web Services Description Language (WSDL) [23], or proprietary interface description languages (IDLs);

- The service broker can implement the standard Universal Description, Discovery and Integration (UDDI) [14], Jini discovery [3], LDAP [10], or other proprietary discovery mechanisms;

- The services can be invoked remotely (the proxy approach) or downloaded to local computer (the applet approach);

- The communication infrastructure among the components and the associated data encoding formats include SOAP [21] and XML [8], RMI [18] and objects, and proprietary RPC standards.

In recent years, Java has emerged as the dominant language for Web programming. Among the standards for Web services, UDDI, WSDL, and SOAP have gained strong industry support, and are likely to become the standard Web services stack for service brokering, description, and invocation, respectively. The following section describes several initiatives for implementing the Web services model and it discusses the standards and technologies supported by each of them.

3. RELATED WORK

With its huge promise to bring Internet-based software development to a new level, it is not surprising that Web services have drawn significant attention from the major players in industry. This section describes the existing efforts for the development and maintenance of Web services, highlighting their core functionality and their limitations.

Microsoft .NET [19] is a framework and a set of developer tools in Visual Studio [16] that share the same goals as TSSuite. The components in the framework communicate through the SOAP protocol. The Rapid Application Development (RAD) tools in Visual Studio facilitate the deployment and maintenance of services. Message passing between components is handled by Microsoft Message Queuing (MSMQ). Although this framework and toolkit is feature rich, a large portion of it is based on proprietary technologies. As of the first beta release, service description using WSDL and the UDDI broker is not supported. Instead, Web services are described using SCL (Service Contract Language) and discovery is handled through a Microsoft's proprietary file-based mechanism called DISCO [15]. In this mechanism, service interface and implementation are not distinguished, which prevents searches for services that implement a given interface. Although service composition is highlighted as a key feature in the .NET overview document, the current framework does not allow users to compose services in a straightforward way.

Sun Microsystems Jini [3] was one of the first available toolkits for services. Jini provides service discovery, remote method

invocation, security, leasing, transactions, and events. It simplifies the deployment and invocation of services and Java applications in a local area network (LAN) environment. Once a Jini service has been discovered, it transmits its interface in the form of a Java class that functions as a proxy. However, Jini pre-dates the widespread acceptance of current Web service standards and, therefore, is not based in any of them. This limits the benefits of Jini for the Web services model, in which interoperability is a key factor.

SunONE [22] is Sun Microsystems "Open Net Environment", a product portfolio that includes tools that support the development of services compatible with UDDI, XML, WSDL and other standards. An example of such a tool is Forte, an integrated development environment for Java that assists service creation. SunONE can be viewed as a peer to Microsoft's .NET or IBM's Visual Age [2] in that its tools are designed to make it simple for developers to construct Web services (or clients to Web services). However, it does not directly address the problems of service composition and deployment, two of the major concerns of TSSuite.

Hewlett Packard's e-Speak [9] is a framework and a set of components for developing Web services. The framework provides support for service creation, deployment, and discovery. The components provide low-level support to Web services, such as message routing and security. E-Speak supports a wide variety of service standards including SOAP, XML, UDDI, J2EE, .NET, and their own J-ESI (Java E-Service Interface). However, e-Speak does not provide an integrated environment for service development and, like SunOne, it currently provides no support for service composition.

Vinci [1], a project at IBM Research, is an infrastructure for developing applications as a set of interacting services deployed over a LAN. It provides support for creating fast and reliable distributed applications through service composition. Vinci prioritizes performance and reliability and, therefore, is not based on any of the Web services standards. Instead, it uses proprietary protocols such as Xtalk [1], a binary representation of XML. The performance experiments described in [1] have shown that services developed with Vinci are considerably faster then the ones based on the standard Web services protocols. An important point is that applications developed with Vinci can still be deployed using TSSuite. In this case they are externalized through standard Web services protocols while benefiting from the high-performance features of Vinci internally.

The Web Services Toolkit (WSTK) by IBM [11] provides a set of tools for assisting the development of Web services and it is compatible with the Web services protocols (SOAP, UDDI, and WSDL). The WSTK tools automate the generation of services from service interfaces described in WSDL (and vice-versa) and allow users to browse and publish information in UDDI. However, WSTK does not provide an infrastructure for composing, deploying, and monitoring services, and it is limited to SOAP-based services. As will be discussed in the following section, TSSuite and WSTK can be used together, once they provide complementary tools.

4. TSSUITE INFRASTRUCTURE AND TOOLS

This section presents the TSSuite design and functionality, showing how its components work together to assist the development and maintenance of Web services. Figure 2 shows the high-level architecture of TSSuite. It includes a UDDI broker, the TSpaces Service API (TSSAPI), and a set of tools. The UDDI broker and the TSSAPI are developed as an extra layer over TSpaces [24], extending its programmatic capabilities towards Web services. The other components are tools for service developers and administrators and can be viewed as clients of the TSSuite infrastructure. The following subsections provide an overview of TSpaces and each of the TSSuite components.
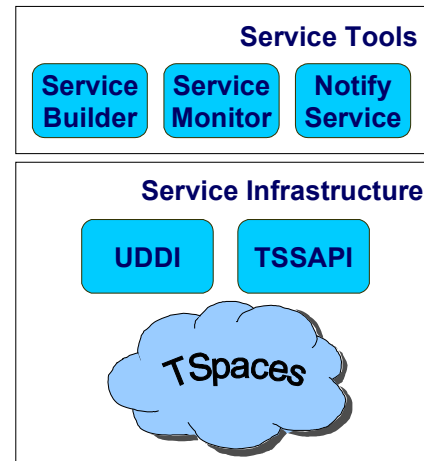

Figure 2. TSSuite main components

4.1 TSpaces overview

TSpaces [24] is a network communication buffer with database capabilities. It enables communication between applications and devices in a network of heterogeneous computers and operating systems using the shared space concept from the Linda programming model [6].
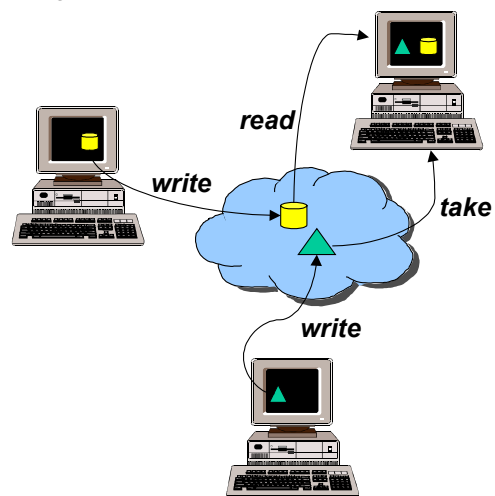

Figure 3. TSpaces basic operations

The basic TSpaces commands are *write, read,* and *take*. The difference between *read* and *take* is that after a *take* the tuple is removed from the space while *read* leaves a copy of the tuple in

the space. TSpaces clients communicate asynchronously through shared spaces by writing, reading, and taking tuples. Each tuple is a vector of Java objects. Figure 3 shows the interaction among three TSpaces clients.

Tuples are not directed to any specific client, meaning that any client can read (or take) any tuple, as long as it has access rights to do so. Tuples reside on the space until someone issues a *take* removing them or until they expire. Therefore, a client can read a tuple even if it, the client, was not alive when the tuple was written. Clients can also register for *event notifications*, such as "let me know when a tuple with the following content is written to the space." When the event occurs (e.g. the specified tuple is written) the client is notified through a *callback* method. Figure 4 illustrates this scenario.
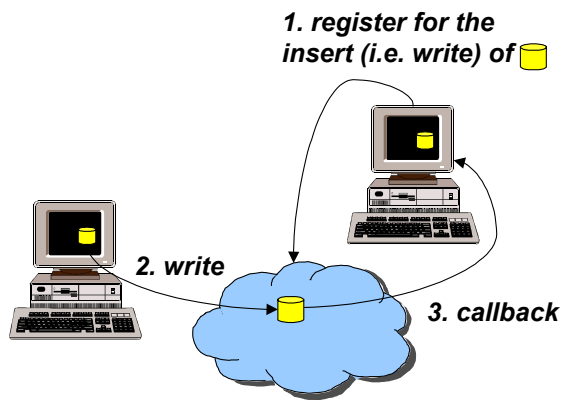

Figure 4. TSpaces event registration mechanism

4.2 TSpaces Service API (TSSAPI)

The TSpaces Service API (TSSAPI) is a framework that simplifies the creation, deployment, configuration, and invocation of services. TSSAPI accomplishes this by standardizing the interfaces between services and by formalizing the types of interactions that services can support. TSSAPI handles all the specifics of the several available communication infrastructures allowing the service developer to concentrate on the service-specific code. Moreover, it frees the service developer from having to deal with administrative tasks, such as service registration.

TSSAPI uses the WSDL vocabulary for service description. The core concepts in TSSPAI are operations, messages, port types, and ports.

An **operation** is a function that the service can perform, such as *convert* in a PDF to PS conversion service (detailed in Section 5). The operation is bound to some object and method that will do the processing as required by the operation. In the PDF to PS example, the *convert* operation is bound to the method *convert()* in the *PDF2PS* class. Operations may be invoked in response to some request, or may start in response to some service-internal or environmental trigger. Operations produce and consume data in the form of messages.

A **message** is a container used to encapsulate data going to or coming from an operation. Messages may contain zero or more parameters (or parts), each having a label and an object-type (String, for example). An operation may contain one input

message, one output message, or one of each (operations containing neither are not allowed). Messages that have no parts can be used for invoking parameter-less service routines (in the case of input messages) or for representing service routines with no return value (in the case of output messages). A **port type** is a collection of operations. Each port type corresponds to a service interface. Figure 5 shows the relationship between port types, operations and messages, in the context of a PDF to PS conversion service which provides two operations: configure and convert.
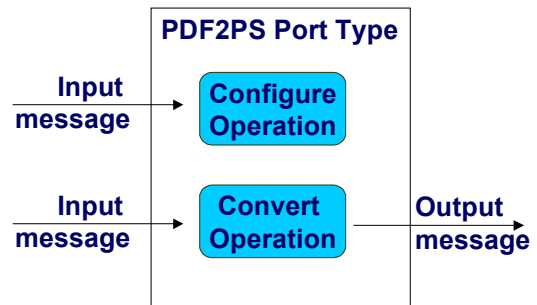

Figure 5. TSSAPI port types, operations and messages

A **port** is an implementation of a port type. Each port is bound to some service communication infrastructure. Ports are also bound to a specific server location, which will define the access points for invoking the service. A port is the specification of the service implementation.

TSSAPI is a flexible architecture that allows additional communication infrastructures through the creation of new sub-classes of the *Port* class. Currently TSSAPI provides implementations for TSpaces and SOAP ports. Services based on the TSpaces port may take the advantage of asynchronous communication, which may be more appropriate for some services[1]. On the other hand, SOAP is a widely accepted standard and is independent of programming language – clients can be written in any language as long as they can send XML messages. The following paragraphs describe the TSpaces and SOAP ports in more detail.
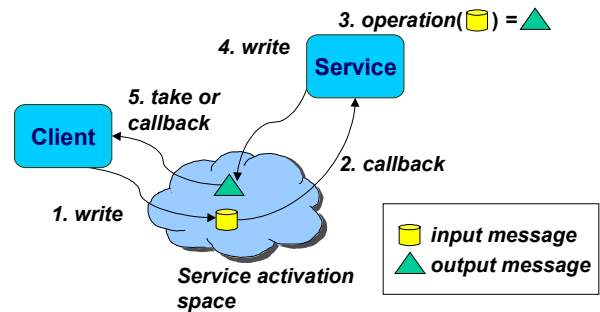

Figure 6. Sequence of events for invoking services through TSpaces ports

---

[1] For instance, a synchronous call to a service operation that does very long computations would block the client for a long time, while an asynchronous call would free the client to do other tasks and it would "call back" the client when the computation is done.

Services deployed in TSpaces ports are invoked by writing tuples into a space, called the service activation space. The result of the operation (output message) is also written to the space. Figure 6 shows the sequence of events that happen when a service is invoked through a TSpaces port. TSSAPI handles the entire interaction among the client, the service, and TSpaces, including the registration for callbacks and the invocation of the appropriate service operations.

In the case of the SOAP port, services are invoked through a SOAP server (TSSuite uses Apache Tomcat [12] as the SOAP server). One important point here is that all the ports in which the service is deployed share the same service code, as shown in Figure 7. Moreover, the service and the infrastructure code are totally decoupled. Even when new service communication infrastructures are defined, TSSAPI does not require any changes to the service code.
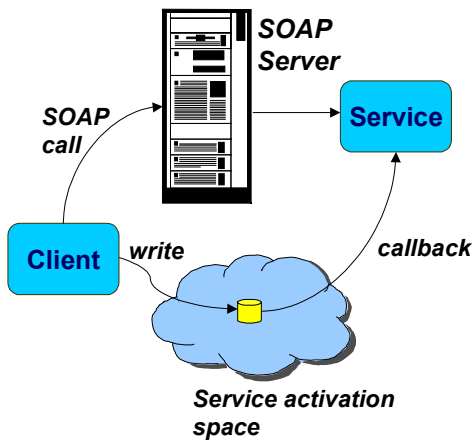

Figure 7. Sharing the service code among different ports

TSSAPI supports service invocation through a library of service proxies. Each port defined in TSSAPI has a corresponding service proxy. Service proxies encapsulate the infrastructure-specific details necessary to invoke services. This is accomplished by a mapping between the port invocation format (e.g. XML in SOAP, tuples in TSpaces) and the TSSAPI standard format. Clients can use the same code to invoke a service in different infrastructures, only changing the line that selects the appropriate proxy. Figure 8 illustrates this design through UML class diagrams.
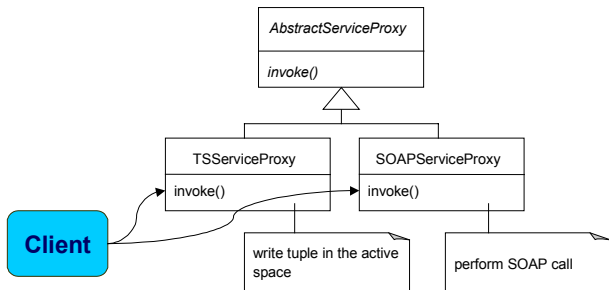

Figure 8. TSSAPI service proxies

Besides the support to service creation, deployment, and invocation, TSSAPI also provides classes for handling service configuration, as described in detail in Section 5.

4.3 TSpaces UDDI Broker

The TSpaces UDDI broker is an implementation of the UDDI specification [14] that uses TSpaces as a global directory for storing Web services information. UDDI is based on SOAP/XML and defines an API for retrieving information about businesses and the services they provide. UDDI is composed of two different API's: publisher and inquiry. The publisher API allows for the registration of businesses and Web services in the global directory. It provides methods such as "save_bussiness" and "save_service." Once this information is registered it can be retrieved through the inquiry API, which provides methods for locating the available services and for retrieving information about how to invoke them. It allows a client to determine dynamically if a service it needs is available, discover how it works, and invoke it. Figure 9 illustrates the UDDI broker architecture. Please note that the UDDI client can be either a service, during service registration, or a client, during service discovery.
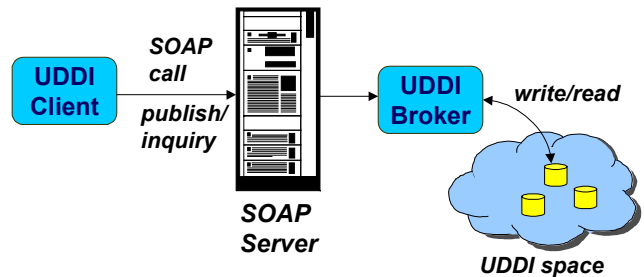

Figure 9. TSpaces UDDI broker architecture

Since all the information published in UDDI is stored in TSpaces, clients can use the TSpaces event registration mechanism to be notified when certain kinds of services are published. A possible scenario would be "let me know when a new bike store is available in California." This form of interaction with the UDDI broker is not part of the UDDI specification and can be viewed as a value-added service. This is a typical example where the TSpaces communication infrastructure is more appropriate than SOAP.

4.4 Service Builder

The Service Builder automates most of the service development and maintenance tasks. It acts as the bridge that connects the service implementation program, the TSSAPI, and the UDDI broker. To deploy a service the user just needs to provide the implementation program and the service interface description, which can be retrieved from UDDI. The Service Builder then extracts the service interface from the WSDL file, finds the Java methods in the service implementation classes, and deploys the service using the TSSAPI service description, deployment, and configuration classes. Figure 10 illustrates a possible interaction between the Service Builder, UDDI, and TSSAPI. Section 5 details how this component is used to assist the development of Web services through a case study.

4.5 Notify Service

The Notify Service (TSSNotify) is a general-purpose mechanism for associating tuplespace events by defining rules that map triggering conditions to actions. TSSNotify was designed to be a low-level component that processes only tuples and tuplespaces. However, since services can use the TSpaces communication

infrastructure, TSSNotify is capable of manipulating standardized service messages just as easily as it can create and route other kinds of tuples. Therefore, TSSNotify can be seen as a tool for integrating (or composing) services based on the TSpaces port.

In this context, the rules specify the control flow among several interacting services, while TSNotify takes care of performing the appropriate service invocations (by writing the appropriate tuples in the service activation spaces). One example is the integration of a PDF to PS file conversion service and a printing service to compose an "intelligent print" service, as shown in Figure 11. The name "intelligent print" is due to the fact that the file conversion will only happen if necessary.
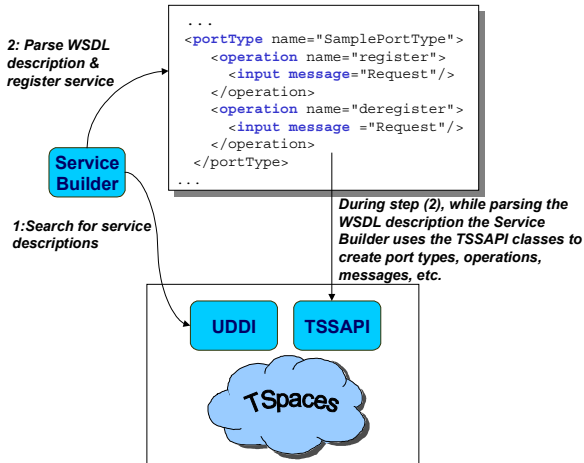

Figure 10. Service Builder interaction with UDDI and TSSAPI

A client activates "intelligent print" by writing a tuple with the file to be printed and the information about the desired printer to the "intelligent print" activation space. Then, TSNotify performs two actions in parallel. It moves file information in the tuple provided by the user to the PDF to PS conversion space and it moves the printer information to the print space. If the file tuple moved is a PDF file, as shown in Figure 11, the PDF to PS service will be automatically invoked and the file will be converted to PS. On the other hand, if the tuple moved is a PS file, steps 3 and 4 will not be performed since the conversion service is invoked only in the presence of PDF tuples. In both cases TSNotfiy will move the PS tuple to the print space, which will automatically trigger the printing process and generate a print report (at that point the printer information is already there, so the print service knows which printer to use). After the report is generated, TSNotify moves it to the "intelligent print" space, generating the client notification. Note that the fact that "intelligent print" is composed of two sub-services is totally transparent to the client. Another important point is that besides the three TSNotify rules, no code was necessary to implement this service. The move rules are easily defined using the TSSNotify user interface explained below. This example illustrates how TSSuite allows the creation of arbitrarily complex services from the composition of simple, basic services.

TSSNotify rules are stored as tuples. This provides several benefits. First, rules that are represented as tuples are easy to duplicate and move from space to space. Furthermore, since TSSNotify operates on rules, it is possible to create rules that, in turn, create additional rules. Finally, rules that are stored as tuples are persistent so even if the host computer crashes or loses power,

the state of the system remains within the space (assuming the TSpaces persistence option is enabled) so that the operation can continue once TSSNotify has been restarted.

TSSNotify provides a rule editor, which is a GUI that makes it easy to create and update rules. Figure 12 illustrates the rule editor interface. On the left side of the panel the user can specify the trigger and action tuples. In the "intelligent print" example the trigger tuple is an event-write tuple, which means that the action will be fired when a tuple that has the specified format is written to the space. The trigger tuple has only one field that holds the file. The action tuple is also an event-write tuple, which means that when the action is triggered a new tuple will be written. In this case the action tuple has also only one field, since the files are simply copied and no extra information is added. This rule specifies the "move" behavior described above, the file tuple is copied from one space to another whenever a file is written to the "intelligent print" activation space. On the right side of the screen the user can specify the rule name and the space in which it will be stored.
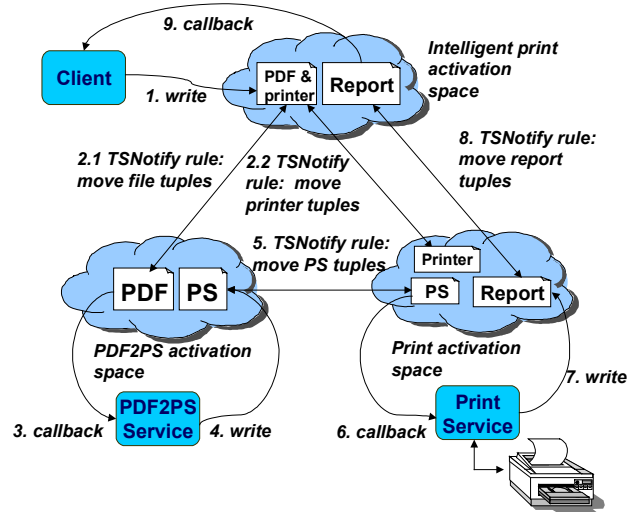

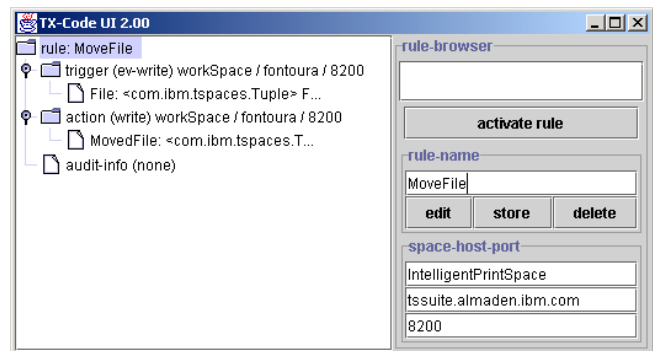**Figure 11. TSNotify application scenario: composing the "intelligent print" service**

Figure 12. TSNotify's rule editor

4.6 Service Monitor
Service Monitor is a visualization tool that works with TSSAPI. It provides a view of service messages and it allows the user to monitor or troubleshoot service logic by viewing the service messages and their contents as they occur.

The Service Monitor can monitor several services simultaneously. Its user interface allows users to selectively add or remove services from the list of services that are being monitored. The user interface also tracks new services and it is aware of services that are no longer present within the system. When a service message for a monitored service occurs, the service monitor displays the message in a table. The table's columns represent services and the table's rows represent messages. Selecting a message from the table displays the message details including the message payload and the time/date the message was received by the server. Figure 13 shows the Service Monitor in action.

Figure 13. Using the Service Monitor to view a list of input and output messages

5. TSSUITE@WORK: THE PDF TO PS CONVERSION SERVICE

This section describes the role of TSSuite in service creation, deployment, and configuration through an example scenario: a file conversion service. Let us assume that we want to offer a Web service that converts files from PDF to PS. In general, we need to go through the following steps to get the service created, deployed, and configured:

1. *Service creation*. The service creation activity is composed of the service implementation (step 1.1) and the specification of the service interface (step 1.2).

1.1 *Service implementation*. For the PDF to PS conversion, we can either write a program that does the transformation or use some an existing application that already provides this feature. A well-known application that can do this conversion is Ghostscript [7]. To take advantage of this, we could simply write a wrapper program in Java that invokes Ghostscript. This program would have a method *convert()* that takes in a PDF file as a parameter and returns a PS file.

1.2 *Interface definition*. If a service is based on an accepted interface or ontology [5], chances are that its interface definition already exists and can be retrieved from a broker. If not, the service interface must be specified in a service description language, such as WSDL.

2. *Service deployment*. The mechanism used to deploy the service depends on the chosen communication infrastructure. For example, if the infrastructure is SOAP, the service interface and the implementation classes must be registered in a SOAP server. After the service is deployed, it should also be registered with a service broker so that potential clients can find it. In general, service deployment is a very mechanical process and should be facilitated by a tool.

3. *Service configuration*. Sometimes a service needs to be configured before it is invoked for the first time. In the PDF to PS conversion service for example, the implementation class has parameters that specify the command used to invoke Ghostscript and a working directory. Parameterization makes the service implementations reusable - once the parameters are not "hard coded" the same service can be used in different contexts with different parameters. One possible solution to facilitate service configuration is to provide a graphical user interface (GUI) in which administrators would enter values for the configuration parameters. However, requiring the service developers to build a GUI for each configurable service would put too much burden on them. In the PDF to PS example, the GUI code would be much longer, and harder to write, then the wrapper code that invokes Ghostscript. Because of this, it is desirable to have the GUI generated automatically by the service deployment tool.

Ideally, a service developer would need to address only step 1.1, which is the actual service implementation. The Web services infrastructure and its surrounding tools should handle all of the other steps, which can be classified as administrative steps. Once the administrative steps have been automated, deploying and configuring a service should be as easy as installing an application on a PC. TSSuite has largely achieved this goal. In the rest of this section we describe the steps we have used to create, deploy, and configure the PDF to PS conversion service with TSSuite.

We first wrote a Java class called PDF2PS.java. This class has a method *convert()* that invokes Ghostscript to do the PDF to PS conversion. To start service deployment, we use the service deployment tool in TSSuite, the Service Builder. Figure 14 shows a Service Builder dialog box asking the user to enter the link to the WSDL service interface and the service implementation class.
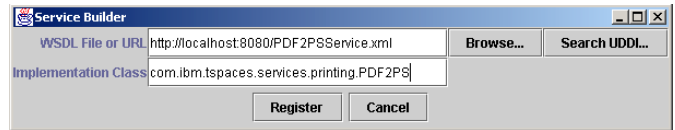

Figure 14. Using Service Builder for deploying the conversion service

For the service interface, the WSDL file can either be provided by the user or retrieved from an UDDI broker. The first case would only be used if the service is new and no interface is available in UDDI. In this situation the Web Services Toolkit (WSTK) can be used to extract the WSDL interface from its implementation classes. Often the WSDL file can be found in UDDI. To perform the search, a partial or full service name, such as "pdf" or "pdf2ps", can be used. Service Builder searches the UDDI broker and it lists all the entries that match the name (Figure 15).

In this example, Figure 15 shows that only one service interface with name "pdf" was registered in UDDI at the time of the search. In a more realistic production environment, the search would likely return multiple entries.

Back in the "Service Builder" dialog in Figure 14, the "Register" button can now be used to start service deployment. The Service Builder will then parse the WSDL description file, find all the operations that must be implemented for this service, use Java reflection to find the methods in the implementation class that implements those operations, and deploy those operations through the TSpaces Service API (TSSAPI). TSSAPI allows the service

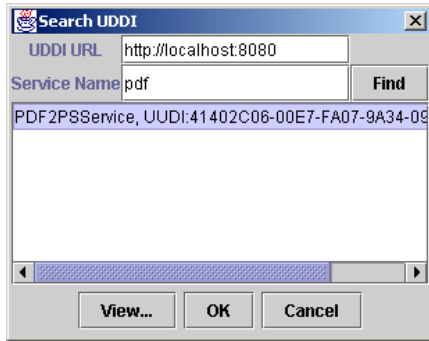developer to deploy a service without having to code for the specifics of each communication infrastructure.

Figure 15. Searching service descriptions in UDDI

TSSAPI also creates a WSDL document for the service. This document is a service implementation (as opposed to a service interface) description, since it describes detailed information about this particular instance of the service, such as the addresses in which the service can be invoked and encoding format for each operation. The service provider can then use the tools in WSTK [11] to register the generated WSDL service implementation file and the WSDL service interface file in an UDDI broker, completing the service registration.

One of the main advantages of WSDL over other service description languages is that it allows the separation of service interface and implementation. This allows service interfaces to be registered in service brokers during service deployment (e.g. as UDDI tModels[2] [14]) and to be searched during service discovery. A search based on the PDF to PS interface, for instance, would retrieve all the service implementations that conform to it.

After passing the service details to TSSAPI, the Service Builder checks if the service needs configuration. It finds this information by checking if the service class implements the *ConfigurableService* interface. This interface is defined as part of the TSSAPI. It has a method *getParameterNames()* that returns the names of the configuration parameters, and a method *configure()* that sets the parameter value. The Service Builder uses this interface to generate the configuration GUI and to configure the service.

In the case of PDF to PS conversion, a configuration GUI that contains the "Conversion command" and the "Working directory" parameters would pop up, as shown in Figure 16. The conversion command is a batch file that sets the path and invokes Ghostscript, the working directory can be any temporary directory. The Service Builder sends the parameter values through the *configure()* method. However, the Service Builder does not call this method on instances of the implementation class directly. Instead, the *configure()* method is treated as a regular service operation. For instance, in the case of a SOAP service the Service Builder would act as a client, invoking the *configure* operation through a SOAP server. Figure 16 illustrates this architecture,

---

[2] A tModel is the UDDI element used to represent services interfaces. Each tModel references one or more WSDL files, and defines all the operations that can be used for a given domain. A tModel for stock quotes would specify operations such as *getQuote, getVolume,* and so on.

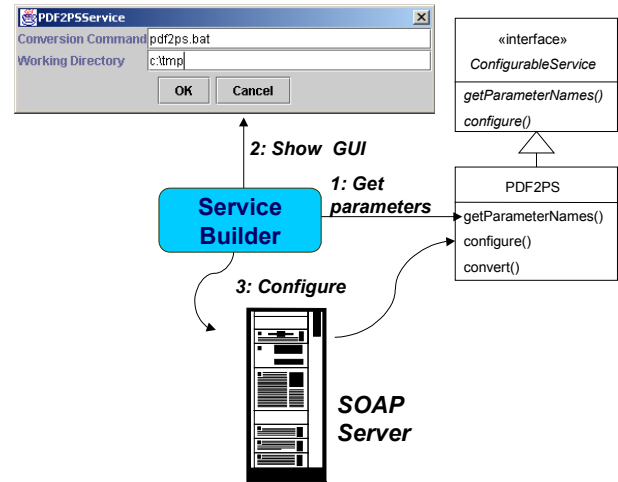through a combination of a UML class diagram and a process description diagram.

Figure 16. Service configuration scenario

As described above, the only component we need to develop to deploy the PDF to PS service is the PDF2PS.java class. Excluding comments, this class has less than 50 lines of Java code and was written in less than an hour. Before the TSuite was available, we also tried to offer this service through SOAP and TSpaces but then the total amount of code we had to write was almost 10 times more.

6. CONCLUSIONS AND FUTURE WORK

This paper described the TSSuite infrastructure and its set of tools. TSSuite automates several administrative activities related to the deployment and maintenance of Web services. Services developed with TSSuite are much more compact (in terms of lines of code) than services developed "from scratch." In the example presented in Section 5 we had almost 10 times reduction in both code size and development time. Section 4.5 has show that TSSuite has been used to develop complex Web services: the TSNotify component allows the composition of basic services to create arbitrarily complex ones, such as "intelligent print". We have also shown how the Web Services Toolkit (WSTK) [11] can be used in conjunction of TSSuite for automating the generation of WSDL files and the interaction with the UDDI broker.

TSSuite is currently available on alphaWorks (http://www.alphaworks.ibm.com/tech/tssuite). In the first months of availability, there have been more than 1,000 downloads. The following table summarizes the code size for each of the TSSuite components. All components have been developed in Java.

Our future research directions include adding fault-tolerance and replication capabilities to the TSpaces infrastructure, which will improve the TSSuite scalability. Another point of investigation is how to map TSSNotify rules to Web Services Flow Language [13] (WSFL) descriptions.

7. REFERENCES

[1] R. Agrawal, R. J. Bayardo Jr., D. Gruhl, and S. Papadimitriou, "Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications,"

Proceedings of the 10th International World Wide Web Conference (WWW'10), 2001.

[2] J. Akerley, N. Li, and A. Parlavecchia, Programming with Visual Age for Java 2, Prentice Hall, 1999.

[3] K. Arnold, B. Osullivan, R. W. Scheifler, J. Waldo, A. Wollrath, and B. O'Sullivan, The Jini Specification, Addison-Wesley Pub. Co., 1999.

[4] R. Englander and M. Loukides, Developing Java Beans, O'Reilly & Associates, 1997.

[5] D. Fensel, Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce, Springer, 2001.

[6] D. Gelernter, "Generative Communication in Linda," ACM Transactions on Programming Languages and Applications (TOPLAS), 7(1), 80-112, 1985.

[7] Ghostscript, Ghostview and Gsview, (http://www.cs.wisc.edu/~ghost/).

[8] C. F. Goldfarb and P. Prescod, The XML Handbook, Prentice Hall PRT, 1998.

[9] Hewlett-Packard Company, "Ten Ways to Think e-Speak," 1999, (http://www.e-speak.net/library/pdfs/ThinkEspeak.pdf).

[10] T. A. Howes and M. C. Smith, LDAP: Programming Directory-Enabled Applications with Lightweight Directory Access Protocol, Macmillan Technical Publishing, 1997.

[11] IBM, "Web Service Toolkit," (http://alphaworks.ibm.com/tech/webservicestoolkit).

[12] The Jakarta Project, "Jakarta Tomcat," (http://jakarta.apache.org/tomcat/).

[13] Frank Leymann, "Web Service Flow Language (WSFL 1.0)," 2001, (http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf).

[14] B. McKee, D. Ehnebuske, and D. Rogers (editors), "UDDI Version 2.0 API Specification," (http://www.uddi.org/pubs/ProgrammersAPI-V2.00-Open-20010608.pdf)

[15] Microsoft, "Enabling Discovery for a Web Service," (http://msdn.microsoft.com/library/default.asp?url=/library/enus/cpguidnf/html/cpconenablingdiscoveryforwebservice.asp).

[16] Microsoft, Visual Studio .NET, (http://msdn.microsoft.com/vstudio/nextgen/default.asp).

[17] R. Orfali, D. Harkey, and J. Edwards, Instant CORBA, John Wiley & Sons, 1997.

[18] E. Pitt and K. McNiff, Java.RMI: The Remote Method Invocation Guild, Addison Wesley Professional, 2001.

[19] D. S. Platt, Introducing Microsoft .NET, Microsoft Press, 2001.

[20] D. S. Platt, Understanding COM+: The Architecture for Enterprise Development Using Microsoft Technologies, Microsoft Press, 2001.

[21] K. Scribner and M. C. Stiver, Understanding SOAP, Sams, 2000.

[22] Sun Microsystems, "Open Net Environment (ONE) Software Architecture," (http://www.sun.com/sunone/).

[23] W3C, "Web Services Description Language (WSDL) 1.1," 2001 (http://www.w3.org/TR/wsdl).

[24] P. Wyckoff, S. W. McLaughry, T. J. Lehman and D. A. Ford, "TSpaces," IBM Systems Journal, 37(3), 454-474, 1998.