WEB FEDERATES - TOWARDS A MIDDLEWARE FOR HIGHLY SCALABLE PEER-TO-PEER SERVICES

Ingo Scholtes and Peter Sturm
University of Trier
Department of Computer Science, Systemsoftware and Distributed Systems, D-54286 Trier, Germany

Keywords: Web Service, Peer-To-Peer, Web Federate, WSDL, SOAP, Middleware, Message Exchange Patterns, Code Generation, Request/Response, One-Way, Solicit/Response, Notification, Publish/Subscribe, Scalability, Light-Weight Hosting, Service Oriented Architecture.

Abstract: Starting from the classical Client/Server paradigm, in the last couple of years Peer-To-Peer approaches have evolved and proven their power. Currently we see an evolution from the distributed object access paradigm represented e.g. by middleware architectures like CORBA, DCOM or RMI towards Service Oriented Architectures (SOA), entailing a retrogression to the Client/Server paradigm. In this paper we want to present how Peer-To-Peer Applications can to a large extend benefit from intrinsic Web Service properties like loose coupling, declarative interface definition and interoperability, thus incorporating advantages from SOA and the Peer-To-Peer approach, opening new fields of application to both of them. For this purpose, WebFederate, a prototype middleware based on Microsoft's .NET Framework has been implemented and will be presented in this paper.

1 INTRODUCTION

Starting from the classical Client/Server paradigm, in the last couple of years Peer-To-Peer approaches have evolved into a serious alternative. Prominent examples like Skype or notorious file-sharing applications but also scientific prototypes like Freenet (Clarke et al., 2000) or Oceanstore (Rhea et al., 2003) show the power of this approach in regard to scalability and availability. While these accomplishments of the Peer-To-Peer paradigm are widely accepted, currently one can see an evolution from classical distributed object access technologies like CORBA, DCOM and RMI towards Service Oriented Architectures (SOA), entailing a retrogression to the Client/Server paradigm. This retrogression is to a large extend based on the idea, that Web Services are hosted by heavyweight Web or Application servers, which automatically results in an asymmetric relationship between caller and callee. Apart from that this deployment model usually implies, that Web Service providers are passive, reacting only to requests from active service consumers. These properties are however no intrinsic characteristics of the underlying standards. Lightweightiness and usability in a decentralized distributed environment are two key design goals which have been respected in the definition of the SOAP standard. Apart from simple Web Service usage scenarios adopted today - Web Services being mainly used as interconnection between back-end servers and presentation front-end - which primarily use the Client/Server paradigm, the SOAP standard defines the SOAP Intermediary role which is suitable for multihop scenarios. These are very likely to be applied when Web Services will need to be deployed in global scale.

Furthermore, the WSDL standard, in addition to the commonly used One-Way and Request/Response message exchange patterns, also defines two others: Solicit/Response and Notification. In most Web Service implementations however, these additional message exchange patterns remain unused for reasons which will be discussed in a separate section.

Regarding declarative service description, loose coupling, late binding and interoperability, Peer-To-Peer applications can to a large extend benefit from these intrinsic Web Service qualities, henceforth incorporating advantages from Service Oriented Architectures and the Peer-To-Peer approach and opening new fields of application to both of them. One fundamental step towards this goal is a light-weight way to create, deploy and host Web Services.

The remainder of this paper is organised in the following way: Section 2 discusses the Solicit/Response

Scholtes I. and Sturm P. (2006).
WEB FEDERATES - TOWARDS A MIDDLEWARE FOR HIGHLY SCALABLE PEER-TO-PEER SERVICES.
In *Proceedings of WEBIST 2006 - Second International Conference on Web Information Systems and Technologies - Internet Technology / Web Interface and Applications*, pages 13-20
DOI: 10.5220/0001253800130020

and Notification message exchange patterns, reasons why these are commonly unused, some code generation technique and related multicast considerations. Having examined scalability issues of the general Web Service technology in section 3, section 4 presents work in progress on a prototype middleware, which remedies Web Services from the aforementioned deficiency of asymmetric role association. Finally, after having done a presentation of related work in section 5, a conclusion will be drawn along with an outlook to future work. For the course of this paper we introduce the term ”Web Federate”, which henceforth denotes a computing node running applications based on this middleware, since these nodes may act in the roles of service provider and consumer simultaneously and swarms of them may federate in order to host superordinate services collaboratively.

2 SOLICIT/RESPONSE AND NOTIFICATION MESSAGE EXCHANGE PATTERNS

There are several reasons why most Web Service implementations do not use the Solicit/Response and Notification message exchange patterns:

- The WSDL-To-Code mapping is more complicated than for the Request/Response or One-Way patterns
- They require service providers to become active, a behavior not foreseen in most container deployment models
- They require some kind of Publish/Subscribe model
- They do not scale in the number of subscribers due to the lack of multicast communication schemes available on the public Internet
- They exhibit concurrency difficulties at the service consumer side
- The most widely used HTTP protocol binding does not make use of Solicit/Response and Notification
- Requirement R2303 of WS-I Basic Profile 1.0 and 1.1 precludes their usage

One reason why these additional message exchange patterns are currently not supported by the WS-I Basic Profile, is that one can describe them also at the consumer’s side in terms of Request/Response or One-Way. Although a Notification operation certainly can be seen as a reverse One-Way operation and a Solicit/Response operation is nothing more than a Request/Response from the callee’s point of view, it does however make sense to use them. In Peer-To-Peer application scenarios we are addressing, a peer may not know what kinds of notifications are available and what it is interested in beforehand. It does however know which notifications it can provide. So it seems natural to describe such a service at the side of the notification provider in terms of a Notification rather than describing it at the service consumer’s side in terms of a One-Way pattern. A similar argument is true for Solicit/Response operations. At the time when a peer needs a service there may e.g. be no appropriate peer providing such a service in terms of Request/Response. The peer searching for a service however does know what he desires thus he can ask for it explicitly by exposing a Solicit/Response operation, so accessory peers which are able to satisfy this desire will learn about it. Admittedly the term ”service provider” may be misleading in this case, as in the case of a Solicit/Response operation, the actual service (in the sense of an attendance) is done by the service consumer. So in many cases the usage of a Solicit/Response service may require some form of altruism on the consumer side in order to use it.

2.1 Code Generation Techniques

As mentioned before, the mapping from WSDL document to proxy and stub code is more complicated for the Solicit/Response and Notification message exchange patterns than for Request/Response or One-Way which can be mapped to (possibly void) methods in a very natural and straight-forward way. Regarding a WSDL document mixing Request/Response, Notification and Solicit/Response message exchange patterns the primary reason for this additional complexity is the fact, that the clear division in proxy and stub code gets lost. At the service provider side, instead of having a stub class with abstract methods to be implemented by the application programmer, due to Solicit/Response and Notification schemes stub classes may now contain additional method implementations that need to be callable by the user. This naturally precludes the container deployment model from being used. Regarding proxy classes at the service consumer side, the deprivation of this division is even more painful, as these become abstract too.
Traditional code generation techniques can however be applied, if middleware tools are allowed to generate several classes for a single service. Instead of the straight-forward approach described above, tools might also generate:

- a traditional proxy class for Notification and Solicit/Response operations at the service provider side
- a traditional stub class for Request/Response and One-Way operations at the service provider side
- a traditional stub class for Notification and Solicit/Response operations at the service consumer side
- a traditional proxy class for Request/Response and One-Way operations at the service consumer side

An example for a Web Service framework which makes use of this code generation technique is LEIF[1], a C++ framework for service-oriented applications which supports all message exchange patterns defined in WSDL, including Solicit/Response and Notification.
Apart from generating proxy and stub code from WSDL documents, state-of-the art middleware also allows automatic generation of WSDL documents from service implementations. In case of the Request/Response and the One-Way message exchange pattern application programmers commonly simply provide a service implementation inheriting a service's base class, tagging methods they want to expose as service operations (e.g. the WebMethod attribute in the .NET context) - the Middleware can then generate the WSDL document along with the code, which actually receives SOAP envelopes, unmarshals the arguments and invokes the methods provided by the application programmer.
Using the Solicit/Response and Notification patterns, the situation is again more complicated as it requires application programmers to provide an abstract class with some abstract methods. Code generation tools would now need to create a class which inherits the user's abstract class and provides implementations for all abstract methods. In these implementations, marshalling of arguments and the actual sending to the subscribers needs to be done. As the subscribers list has to be known in order to actually send notifications and soliciting requests, there need to be means of making this list available to the middleware.
Comparing the aforementioned abstract service class from which middleware shall generate a WSDL document and the stub code that would have been generated from the same WSDL document, `abstract` qualifiers are transposed. Actually this transposition becomes evident, if one looks at an `abstract` qualifier as a "request for implementation" for this method. Starting with a WSDL description, in the case when a stub class has been generated from the WSDL document, the middleware requests the application programmer to provide service operation implementations it can not provide itself. In the latter case, starting with the implementation, a WSDL document being generated from it, the application programmer requests the middleware to provide marshalling operations he does not want to provide himself.

[1] http://www.roguewave.com/products/leif/

2.2 Achieving Multicast Communication

As the public Internet lacks true multicast communication schemes, the sending of Notification and Solicit/Response messages to a large number of subscribers does not scale. One possible solution to overcome this problem in case of the Notification message exchange pattern is the usage of distribution trees, playing on the federate role of the subscribers. Instead of shipping a Notification message to all subscribers, the service provider sends it to a constant number of federates, these redistributing messages in an overlay distribution tree. This allows the shipping of notifications to all subscribers with only a constant number of subscriptions per federate although it induces additional latency depending on the position of a federate in the tree. A similar scheme has been implemented in the ATLAS Event Monitoring System, which will be used in order to distribute collision event data among physicists' monitoring processes in a scalable manner at CERN's forthcoming LHC particle accelerator experiment. A more detailed discussion and evaluation of this content distribution scheme can be found in (Scholtes, 2005) and (Kolos and Scholtes, 2005).
In case of Solicit/Response message exchange patterns the solution is however not that simple, as response messages have to be routed back to the service provider, thus leading to an implosion problem. Data aggregation techniques may find application along the path in order to solve this problem, presumably requiring domain-specific knowledge.

3 WEB SERVICE SCALABILITY

Today mostly major companies like Amazon, Google or eBay are involved in large-scale deployment of Web Services and a lot of time and effort is being put into making them scale. Although vulnerability for attacks and sabotage as well as laborious scalability and fault tolerance are intrinsic properties of centralised Client/Server architectures, for the reasons set forth in section 1 it is the one which is most widely spread in the context of Web Services. In his recent article (Birman, 2005) Ken Birman argues, that in the future all sorts of companies and organisations will be interested in deploying Web Services at a large scale. Many of those will not have the means Amazon, eBay

or Google have.
While also big players might benefit from Peer-To-Peer approaches, relieving them from cost-intensive scalability and fault tolerance efforts, Peer-To-Peer technology is likely to become inevitable for smaller institutions willing to deploy Web Services at large scale. The success of a small start-up company like Skype[2] could hardly have ever been imagined if rather than relying on Peer-To-Peer technology they had had to set up cost-intensive network infrastructure in the first instance. Similar scenarios are also imaginable for Web Service technologies. Furthermore, regarding a scenario of Web Services becoming the primary communication architecture across the Internet, even individuals might want to offer scalable Web Services. Today, Weblogs are one example of Web Service technology that is already used by a still-growing number of millions of individuals. While currently for example RSS feeds are implemented using pull model technology simulating a push model behavior, technologies like Trackback also today make use of a push model similar to WSDL's Notification communication scheme, requiring Weblogs of individuals to offer and consume Web Service in a federate manner. Taking into account the tremendous increase in bandwidth, computing power and storage capacity observable in home computing devices over the past couple of years, today the application of Peer-To-Peer technology appears more promising than ever. Future technologies like network filesystems similar to prototypes like Oceanstore (Rhea et al., 2003) but also mobile devices with limited storage and computing power are therefore very likely to rely on Peer-To-Peer technology and the means provided by Web Services.
Apart from mere scalability, in the future Web Services might find application in critical domains which require a high level of immunity against failures and deliberate breakdowns. Regarding scalability and decentralism being intrinsic properties of Peer-To-Peer approaches and interoperability being a key characteristic of Web Services, it seems to be evident that a marriage between these techniques is about to happen. Power-law obeying overlay networks similar to those used in today's Peer-To-Peer applications might be utilised in the future to create swarms of Web Federates providing superordinate services in a fault-tolerant, self-organising and scalable way. For this purpose means like those introduced in (Devlin et al., 1999) seem to be appropriate.

[2]http://www.skype.com

4 WEBFEDERATE MIDDLEWARE

In order to empower developers to achieve the goals presented above, a prototype middleware based on Microsoft's .NET Framework and Microsoft's Web Services Enhancement Toolkit has been implemented, which provides the following key functionality:

- Lightweight and stateful in-process hosting of Web Services
- Lightweight and stateless container model hosting of Web Services
- Dynamic Web Service discovery, proxy-generation and invocation at run-time
- Dynamic Composition of Web Services at run-time
- Automatic exposition of a simple discovery service on start-up
- Automatic service discovery of Web Federates accessing a service

4.1 Overall Architecture

The middleware prototype consists of three libraries: WebFederate, WebFederate.Visualization and Cassini. The purpose of these libraries will be described in the following paragraphs.

WebFederate Visualization Library This library may be used in order to visualise Web Federates along with the services they expose in a graphical manner. It is used by the example application presented in 4.2 in order to illustrate the middleware's dynamics and allow GUI driven interaction with Web Services and Web Federates.

WebFederate The public application programmer's interface of the library's main class `WebFederate` is shown in figure 1. Lightweight Web Service hosting is achieved by using a library which has been implemented on top of Microsoft's shared-source Cassini project, Web Service hosting capabilities offered by ASP.NET and the in-process Web Service hosting provided by Microsoft's Web Service Enhancements (WSE3). Web Services are published using SOAP's TCP and/or HTTP binding according to the application programmer's settings.
The Cassini project is an easy-to-use Web Server library with a small memory footprint published as a shared-source project by Microsoft[3]. Along with ASP.NET it is used for lightweight and stateless

[3]http://www.asp.net/Projects/Cassini/Download/Default.aspx

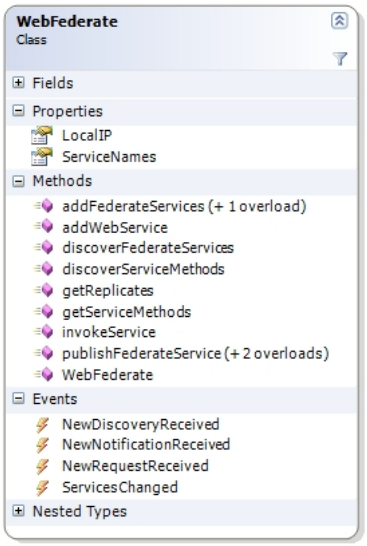

Figure 1: The WebFederate API.

Web Service hosting following the container model and using HTTP as transport. Similar lightweight hosting approaches can be found in the UNIX world e.g. using combinations of lightweight Web Server implementation like abyss [4] or NanoHTTP [5] and the eSOAP[6] toolkit. As ASP.NET Web Service hosting follows the container model using a per-request instance creation, it is per-default stateless leaving state management to the application programmer.
Stateful Web Service hosting is provided by using Microsoft's Web Service Enhancements (WSE), which in this regard pretty much allows a glance at Microsoft's upcoming service-oriented Communication Foundation (codename Indigo). Rather than creating object instances on a per-request basis as done by ASP.NET, instantiation is done on application start-up, the lifetime of object instances being managed by the WebFederate middleware. Federate Services using this hosting facility are therefore inherently stateful. Apart from TCP and HTTP which can be used as transport mechanisms according to the application programmer's preference, the usage of a UDP transport for true asynchronous communication schemes is at the planning stage.
The development procedure commonly used for Web Service implementations and Web Service consuming applications is, even though WSDL offers declarative service descriptions, to a large extend still influenced by distributed object access middleware like CORBA or DCOM. Stub and skeleton classes are generated from a WSDL service description by WSDL parsers (like the `wsdl` tool in Visual Studio, `WSDL2Java` tool in AXIS or the `leifgen` tool in LEIF) at - or actually even before - compile-time, functionality then being added to these generated stubs and skeletons by the application programmer, just like it used to be done with traditional IDL compilers in the distributed object access middleware approaches mentioned above. In order to allow dynamic invocation, creation of and binding to Web Services at run-time, the WebFederate middleware presented in this paper makes use of the declarative service description and .NET's reflection and run-time compile facilities. At run-time proxy code will be generated from the WSDL description of a Web Service, the resulting proxy code will be dynamically compiled and invoked in the background. A wrapper class offers dynamic access to the Web Service methods, which are queried from the generated proxy via the reflection mechanism. All of the above steps are done transparently for the application programmer, so all he has to specify is the WSDL document's URI or the IP address of the Web Federate machine along with the name of the service, enabling him to obtain an object instance at run-time which he may utilise in order to use a service.
In order to discover Web Services running at a Web Federate computing node, on installation-time a straightforward discovery Web Service is automatically created, which in the prototype simply returns a list of all services hosted by a Web Federate. This list can then be used by the requesting Web Federate application in order to dynamically create a specific Web Service proxy as described above. As this discovery service is exposed using a preassigned service name, it can easily be accessed by other Web Federate applications. In emphasis of the symmetric role of Web Federates, hooks have been embedded into the discovery service which will trigger an event whenever a Web Federate performs a discovery request, so the users of the middleware can immediately discover and make use of services provided by the calling federate.
The prototype middleware offers a dynamic deployment mechanism of Web Services. C# Code can be submitted to the middleware, which will try to compile, instantiate and publish the service immediately at run-time. For this purpose a class library along with an `asmx` file is produced, providing the WSDL description of the new Web Service. The service itself is then published by instantiating the service class and registering it with Microsoft's WSE runtime. In order to simplify service development, an abstract class `FederateService` may be used as base class for custom services, it's public interface being shown in figure 2. The high-level architecture of the middleware can be seen in figure 3. Apart

[4] http://www.aprelium.com
[5] http://www.cwc.oulu.fi/nanoip/
[6] http://esoap.ultimodule.com

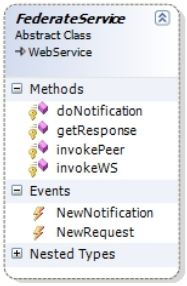

Figure 2: The FederateService interface.

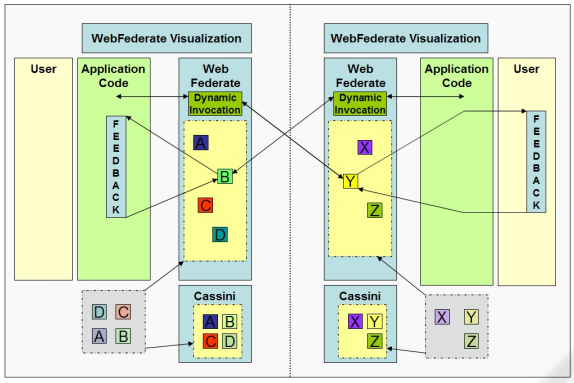

Figure 3: Middleware Architecture.

from traditional service implementations, application programmer's may use the abstract service base class `FederateService` for dynamic Web Service implementations which call back application code on request. The connection between Web Service implementation and application code is achieved making use of .NET's delegate/event mechanism. The implementer of applications based on the middleware may register event handlers dynamically which will be called on reception of requests. The application code handler can then satisfy the request itself or even pass it on to the user.

4.2 Example Application

On top of the middleware presented in chapter 4 a simple graphical test application, the "Web Federate Demonstrator" has been implemented. While this application does not serve a special purpose justified by practical needs, it has mainly been implemented in order to test the middleware and demonstrate how dynamic Web Services can be implemented and deployed using it. A screen shot of this application can be seen in figure 4.

In the address box, a user of this application may enter the URI of a Web Federate or a conventional Web Service's WSDL document. If a WSDL document's address has been entered, a graphical symbol for this Web Service will appear in a panel, if a Web Federate's IP address has been entered a service discovery will be performed, showing all available services on the selected Federate. The user may then select services and add graphical symbols for them to the panel. In order to interact with Web Services added earlier, users may add so-called interface nodes to the panel, connecting them to Web Services dynamically and graphically for invocation using the above-mentioned `WebFederate.Visualization` library. Web Services can simply be published by pushing a single button and completing a service skeleton based on the `FederateService` class. When an interactive Web Service has been published by the Web Federate node and a request to this service has been performed which requires a user's feedback, a window pops up, asking for the user's response and delivering this response to the caller. Using the Web Federate Demonstrator application, it is a question of only a minute to implement e.g. a simple chat application at run-time in a graphical manner.

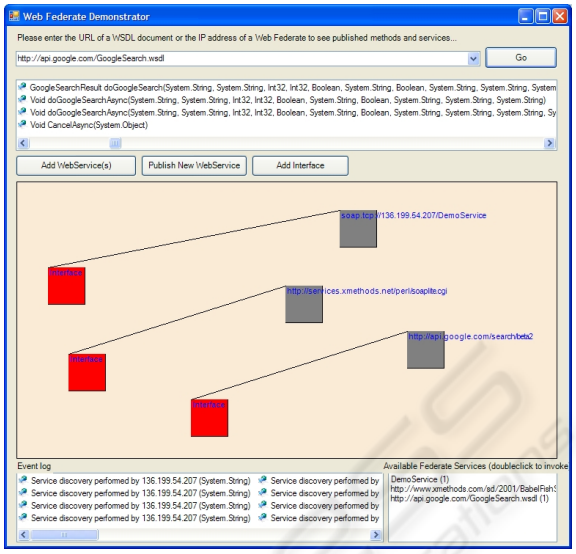

Figure 4: Screen shot of the Web Federate Demonstrator Application.

4.3 Usage Scenarios

In this section some example usage scenarios for middleware approaches like WebFederate will be presented.

E-Learning Although not being the original intention, the "Web Federate Demonstrator" application

presented in section 4.2 can be used right-away as an E-Learning application for introductory programming courses. One might think of a scenario where the lecturer assigns a task to his students, every student using a computer running the "Web Federate Demonstrator" application. In order to solve the task a Web Service has to be implemented at a specific Web Federate which provides a certain given functionality. It is the student's job to collaboratively compose this service possibly combining publicly available Web Services and services they need to implement themselves just as described in the above paragraph. Taking into account the wide variety of Web Services which are publicly available at no charge e.g. listed by (Fan and Kambhampati, 2005), comprehensive and expedient yet feasible tasks can be easily defined. This scenario boosts the student's domain decomposition, teamwork and programming skills and it early introduces collaborative thinking, distributed programming and the RPC paradigm without needing knowledge about the technical details this usually requires. The early introduction of Web Service programming in CS1/CS2 classes has been earlier proposed by (Lim et al., 2005).

Scalable Web Services As pictured in section 3, in the future Peer-To-Peer approaches will become more important in the context of Web Services. This requires special middleware architectures which are capable of providing a self-organisational and scalable infrastructure. The WebFederate middleware represents one step towards this goal as it provides basic techniques for light-weight Web Service hosting and dynamic invocation.

Ubiquitous Computing Light-weight Web Service hosting techniques are crucial for ubiquitous computing scenarios, regarding small devices with their very limited processing power and memory layout. Although the WebFederate middleware is based on Microsoft .NET and provides Web Service hosting with a very small memory footprint, it can not be used for .NET enabled mobile devices right-away as it relies on the hosting capabilities of the ASP.NET framework, which are not included in the .NET Compact framework available for these appliances. In the future, investigation shall be done how to overcome this deficiency as the availability of such a middleware for mobile devices might enforce the emergence of ubiquitous computing applications.

5 RELATED WORK

(Harrison and Taylor, 2005a) and (Harrison and Taylor, 2005b) introduce WSPeer, a JAVA-based Web Service middleware for Peer-To-Peer applications. Just like the middleware presented in this paper, in WSPeer deployment of Web Services does not follow the container model or require a Web Server. As it relies on the classic RPC paradigm, it does however not support advanced message exchange patterns like Solicit/Response or Notification. Regarding ubiquitous computing application scenarios, some research on light-weight means of Web Service hosting for embedded devices has been made by (Pratistha et al., 2003), proposing the Micro-Services framework. EIRI, a JAVA based approach for a lightweight Web Service deployment framework from the year 2002 as presented by (Gergic et al., 2002), does not make use of standards like SOAP or WSDL and therefore lacks interoperability. Today however the term Web Service is commonly associated with these standards. Besides that, asymmetry is an intrinsic feature of the framework, as it has been designed in order to support information retrieval of light-weight devices like PDAs, telephones or information terminals from heavy-weight back end database systems. Apache's Web Service Invocation Framework (WSIF) and the AXIS toolkit from the Java world allow dynamic proxy generation and invocation of Web Services. Furthermore WSIF supports Solicit/Response and Notification message exchange patterns at the service consumer side.

6 CONCLUSION AND FUTURE WORK

In this paper we argue that Peer-To-Peer scenarios can be of great use for Web Services. In order to investigate means for light-weight Web Service hosting based on currently available techniques, a simple and generic Peer-To-Peer middleware has been implemented and - along with some use cases - presented. In a future version of this middleware the usage of SOAP intermediaries for the creation of highly scalable and self-organising Web Service Federate networks which act like a single superordinate service provider will be investigated. It seems to be evident, that forthcoming technologies like Microsoft's Windows Communication Foundation middleware (codename Indigo) will boost the emergence of Web Federates, as they will propagate simple-to-use and light-weight techniques for in-process Web Service hosting. While it is common to look at Web Services as the next generation distributed object access, SOAP being a new declarative RPC protocol, we think that this reduction to the RPC paradigm is a fatal underestimation of the standard's potential. RPC-like communication schemes provide only a small portion of the Web Service standards' capabilities and - as they often make assumptions about the underlying implementation - even are counterproductive regarding the

original intention of Service Oriented Architectures. An important fact presented in this paper is, that Web Services are not inherently associated with the Client/Server paradigm, just because they are commonly hosted by Web or Application servers in a traditional 3-tier architecture that separates program logic, data and presentation. In order to build highly scalable systems, Peer-To-Peer approaches to Web Services are required and can actually be implemented using state-of-the-art techniques as proven by the WebFederate middleware prototype.

REFERENCES

Birman, K. P. (2005). Can Web Services Scale Up? *IEEE Computer*, 38(10):107–110.

Clarke, I., Sandberg, O., Wiley, B., and Hong, T. W. (2000). Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66.

Devlin, B., Gray, J., Laing, B., and Spix, G. (1999). Scalability terminology: Farms, Clones, Partitions, Packs, Racs and Raps. *CoRR*, cs.AR/9912010.

Fan, J. and Kambhampati, S. (2005). A snapshot of public Web Services. *SIGMOD Record*, 34(1):24–32.

Gergic, J., Kleindienst, J., Despotopoulos, Y., Soldatos, J., Patikis, G., Anagnostou, A., and Polymenakos, L. (2002). An approach to lightweight deployment of Web Services. In *SEKE*, pages 635–640.

Harrison, A. and Taylor, I. (2005a). Dynamic Web Service Deployment Using WSPeer. In *Proceedings of 13th Annual Mardi Gras Conference - Frontiers of Grid Applications and Technologies*, pages 11–16. Louisiana State University.

Harrison, A. and Taylor, I. (2005b). WSPeer - an interface to Web Service hosting and invocation. In *HIPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models*. To be published.

Kolos, S. and Scholtes, I. (2005). Event Monitoring Design. Technical report, CERN.

Lim, B. B. L., Jong, C., and Mahatanankoon, P. (2005). On integrating web services from the ground up into CS1/CS2. In *SIGCSE*, pages 241–245.

Pratistha, I. M. D. P., Nicoloudis, N., and Cuce, S. (2003). A micro-services framework on mobile devices. In *ICWS*, pages 320–325.

Rhea, S. C., Eaton, P. R., Geels, D., Weatherspoon, H., Zhao, B. Y., and Kubiatowicz, J. (2003). Pond: The OceanStore prototype. In *FAST*.

Scholtes, I. (2005). A reimplementation of the CORBA-based Event Monitoring System for the ATLAS LHC Experiment at CERN. Diploma thesis, University of Trier.