ACTIVE AETHER

WEB SERVICES

Robert F. MacInnis, Ph.D.
June 2017

AetherWorks
501 Fifth Avenue
New York, NY 10017

E: info@activeaether.com

1 WEB SERVICES MODEL

Section 1.1 begins by describing the Web Services reference architecture and introduces the primary entities involved in all Web Service interactions. It continues with an overview of the key Web Services standards of SOAP, WSDL, and UDDI, and concludes with a discussion of efforts to create standardized Web Service interaction guidelines. Section 1.2 describes the Web Service actors of Service Provider, Discovery Service, and Service Consumer. It details the lifecycle activities consumed by each actor, their roles and responsibilities, and examples of technologies involved at each stage of the Web Service lifecycle. Section 1.3 concludes the discussion of the traditional Web Services model with an introduction to implementing Web Services using existing Web Services as building-blocks – describing, packaging and deploying them like components in an assembly in a process termed 'Web Service Composition'.

1.1 Web Services Architecture

The standards introduced by Web Services provide the means to evolve distributed systems from tightly-coupled distributed applications into loosely-coupled systems of services. This evolution has lead to the development of Service Oriented Architectures (SOAs), of which Services are the primary component. The guiding characteristics of SOAs are the interoperation between loosely coupled autonomous components, the promotion of code reuse at a macro (service) level, and architectural composability.

The W3C Web Services Architecture Working Group (WS-Arch) [1] defines a reference architecture for Web Services in [5] which provides a framework for interactions between participants in a service-oriented distributed system comprised of Web Services. The reference architecture describes interactions between loosely-coupled, platform-independent systems based on Web Service standards.

1.1.1 Interaction Model

The WS-Arch specification introduces the three primary entities in Web Service interactions: the 'Provider Entity' (Service Provider), the 'Requestor Entity' (Service Consumer), and the 'Discovery Service'. The Service Provider is responsible for deploying a Web Service and making it available to Service Consumers. Discovery Services provide a searchable directory of Web Service endpoints. Service Providers use a Discovery Service to publish information about active Web Service endpoints. Service Consumers can use a Discovery Service at design time to discover and write programs against Web Service endpoints. Service Consumers can also implement code to use a Discovery Service at run-time to dynamically locate endpoints

(discussed later in section 2.1). The diagram in Fig. 1 below demonstrates the interaction between service providers, service consumers, and a discovery service.

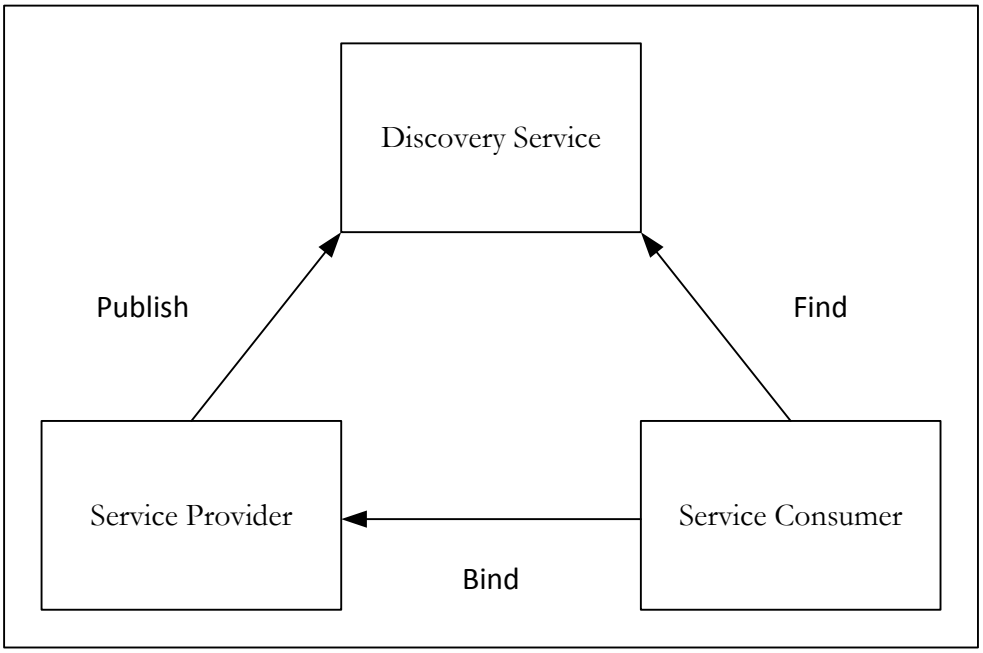

Fig. 1 The Ws-Arch Web Service interaction model.

Individual application components exposed as Web Services are called 'provider agents' – as distinct from the 'provider entity' responsible for providing them (such as a person or organization). Similarly, 'requestor entities' use 'requestor agents' (client-side applications) to exchange messages with a provider entity's provider agents [5]. The terms Service Provider and Service Consumer are often used ambiguously to refer to both entities and agents, and it is important to clarify this distinction.

The remainder of this document will use the term Service Provider to refer to the provider entity; the executable code implementing the provider agent will be called 'service code'; deployed service code exposed as a Web Service will be called an 'endpoint'. Because the requestor entity uses a requestor agent explicitly (rather than just being responsible for it) the term 'Service Consumer' will refer to both the requestor entity and the requestor agent together except where explicitly noted.

1.1.2 Web Services Description & Communication Standards

The following sections introduce the three core standards of the Web Service architecture which provide the means to describe (WSDL), advertise & discover (UDDI), and communicate (SOAP) with Web Services.

1.1.2.1 SOAP

Web Services interact through the exchange of messages using SOAP[1], a protocol designed for exchanging structured information in a decentralized, distributed environment [2]. SOAP defines a communication protocol for Web Services which is independent of programming languages and platforms and is designed to be used over a broad range of transport protocols. It is designed to be both simple and extensible. Simplicity is addressed in the messaging framework by specifying only the message constructs, without mention of common distributed system features such as reliability, security, or routing path. Extensibility is addressed by using XML to describe messages, enabling further specifications to define features which can be added to messages while preserving their interoperability.

SOAP messages are comprised of an outer XML element called `envelope` which defines the namespace(s) for the message, an optional `header` element which includes any relevant extensions to the messaging framework, and a required `body` element. The `body` element provides a 'mechanism for transmitting information to an ultimate SOAP receiver'[2] but is left intentionally unspecified beyond this role, with neither a defined structure or interpretation, nor a means to specify any processing to be done.

SOAP Messages may be exchanged over a variety of underlying protocols conforming to the SOAP Protocol Binding Framework [2]. SOAP over HTTP is used as the reference protocol binding definition and is universally adopted as the primary transport protocol for Web Services [4].

1.1.2.2 Web Service Description Language (WSDL)

The Web Service Description Language (WSDL) is the industry standard language for describing Web Services in XML [6]. The ComponentInterfaceDescription of a service is realized as a WSDL document, which also includes all of the information required to locate, bind to, and interact with a Web Service endpoint.

[1] *Commonly thought to mean 'Simple Object Access Protocol', SOAP is simply a name, not an acronym [2].*

WSDL documents are constructed from XML document elements that describe Web Service endpoints in terms of their operations, the parameters and return values of each operation (including type definitions), and the protocol and data bindings used for communication. A set of operations is called a '*Port Type*' in WSDL and the protocol and data bindings are together called a '*Binding*'. An active Web Service endpoint is called a '*Port*' and is a combination of a binding and a static physical network address where it can be contacted (realized as a URL).

Programmers write requestor agent software against an endpoint's WSDL document. While possible to do by hand, programmers commonly use automatic code-generation tools to generate proxy or 'stub' code in their language of choice. The broad uptake and acceptance of Web Services has led to the development of a variety of such tools for generating code in a multitude of programming languages. These can be stand-alone tools, such as the popular 'WSDL to Java' [7] for Java, as well as IDE-based, such as in VisualStudio [8] and Eclipse [9], which provide a pluggable framework for generators producing code in Perl, C++, C#, PHP, and others.

1.1.2.3 Universal Description Discovery and Integration

Universal Description Discovery and Integration (UDDI) [10] is a Web Service standard for the Discovery Service. UDDI is developed by the OASIS consortium [11] and is based on technology originally donated by IBM, Ariva, and Microsoft. UDDI realizes the Discovery Service functionality as a searchable directory providing the means to describe, advertise, and search for information about Web Services. The standard specifies a service description format and a set of APIs through which entities can interact with the directory.

Service Providers interact with the UDDI directory through the 'Publishers' API which provides operations for adding, modifying, and deleting information about the Web Services they provide. This information includes details about the provider entity, the interfaces implemented by the Web Service (called tModels), and any other proprietary information deemed relevant to a Service Consumer (such as an endpoint location, although this is not a requirement). The Service Provider completes their publication task by storing this information in a UDDI directory under one of the three following categories:

- **WHITE PAGES** list organizations and the services they provide.
- **YELLOW PAGES** classify organizations and Web Services into standard or user-defined groups (such as 'pizza delivery' or 'payroll services').

- **GREEN PAGES** augment the standard Web Service information with pointers to service-description documents detailing how the Web Service can be invoked.

Service Consumers interact with the directory through the 'Inquiry' API which provides operations to search the various directories for Web Service information published by Service Providers. Programmers may use a UDDI directory at design time to find appropriate Web Services for their application. The resultant requestor agent software may also use the directories for run-time dynamic binding.

Reliability and performance have been widely identified as problems inherent in the UDDI registry design (e.g. [12] [13] [14] [15]). In terms of reliability and performance, the two Service Consumer usage scenarios above (static and dynamic binding) present two separate sets of requirements for UDDI registry implementations [16]: while design-time lookup has limited requirements for both reliability and performance, optimizing both of these characteristics is crucial to effective run-time binding [17] [18]. Most work addressing the scalability and reliability of UDDI registries has focused on the use of peer-to-peer (P2P) systems [19] [20] [21] [22]. Although UDDI does not require published information to be valid, P2P technologies have also been applied in [23] and [24] in an effort to maintain accurate, up-to-date, and reliably stored information about Web Services.

A full taxonomy of approaches to improving the reliability and performance of the Discovery Service (including an evaluation of UDDI's competing standard called ebXML [25]) is presented in [26]. Addressing the clear drawbacks of using UDDI registries, novel alternative approaches to providing the Discovery Service are presented in section 2.2 'By Discovery Services' and in 'A Next Generation Architecture For Web Services On The Internet'.

1.1.3 WS-I Interaction Profiles

The standards introduced by Web Services provide a flexible and extensible framework for service interactions over the Web. This freedom is necessary to ensure that Web Services can adapt to and absorb future technologies, and thus the specifications lack any concrete requirements or guidelines as to how Web Services must communicate and behave. Products for creating, deploying and interacting with Web Services may utilize an infinite number of configuration options while still conforming to the Web Services specifications. Using the term 'Web Service' to describe two different technologies which are not interoperable (due to differing version numbers, transport protocols, character sets, etc.) makes it difficult

for users and developers to piece together a Web Service application which is compatible with other services and products.

The Web Services Interoperability Organization (WS-I) [27] provides guidelines for developing and identifying Web Services which are interoperable. It provides a set of 'profiles' dictating the Web Service standards, the specific revision numbers of those standards, and the communication patterns supported by profile-compliant clients and services.

The first WS-I interoperability profile, called the WS-I Basic Profile [28], requires conformance with WSDL 1.1 [3], UDDI 2.0 [29], and SOAP 1.1 [30]. The profile has been widely adopted and is has become the first ISO standard for Web Services interoperability (ISO/IEC 29361:2008) [31].

1.2 Web Service Actors: Roles & Responsibilities

The follow sections detail the roles and responsibilities of the three actors in the Web Service interaction model: Service Provider, Discovery Service, and Service Consumer. Each section outlines the generic tasks undertaken by each together with examples of supporting technologies.

1.2.1 The Service Provider

A Service Provider is a person or organization that is offering a Web Service [5]. Service Providers are responsible for deploying service code as Web Service endpoints which are capable of performing the operations associated with a service. Service Providers are responsible for administering the target environment and consume all of the responsibilities of the Planner and Executor deployment roles. Service Providers acquire service code, create a deployment plan, prepare the target environment, and order the execution of service code. They are responsible for describing the resultant endpoint in a WSDL document and may publish its existence using a Discovery Service.

Service Providers instantiate Web Service endpoints on target environment nodes by executing service code in environments commonly referred to as 'containers'. Containers are software applications which provide access to the business logic implemented by the service code [32]. Web Service containers accept and translate platform-independent SOAP (from the Service Consumer) into the required language-specific data types before the request is processed by the business logic of the requested service. Any return results are translated back into SOAP-format and returned to the Service Consumer.

Web Service containers exist for a wide variety of programming languages. Apache Axis [33] for Java is the most popular stand-alone container, however containers more often represent the 'application layer' of a larger 'application server', as shown in Fig. 2. Application servers provide a Web-accessible presentation layer which coordinates access to the application layer. The application layer is responsible for executing service code and providing that code with access to a resource management layer through standard APIs (such as JDBC [34] and ODBC [35]). Software vendors use application servers to provide the presentation layer and execution environment for a wide array of applications which use the document as the basic unit of transfer [36]. For example, commercial application servers such as Sun One [37], IBM Web Sphere [38], Microsoft .NET [39] and BEA WebLogic [40] are all capable of serving both static and dynamic content to web browsers, integrating with SMTP servers for delivering email over the Web, and all support the deployment of Web Services.

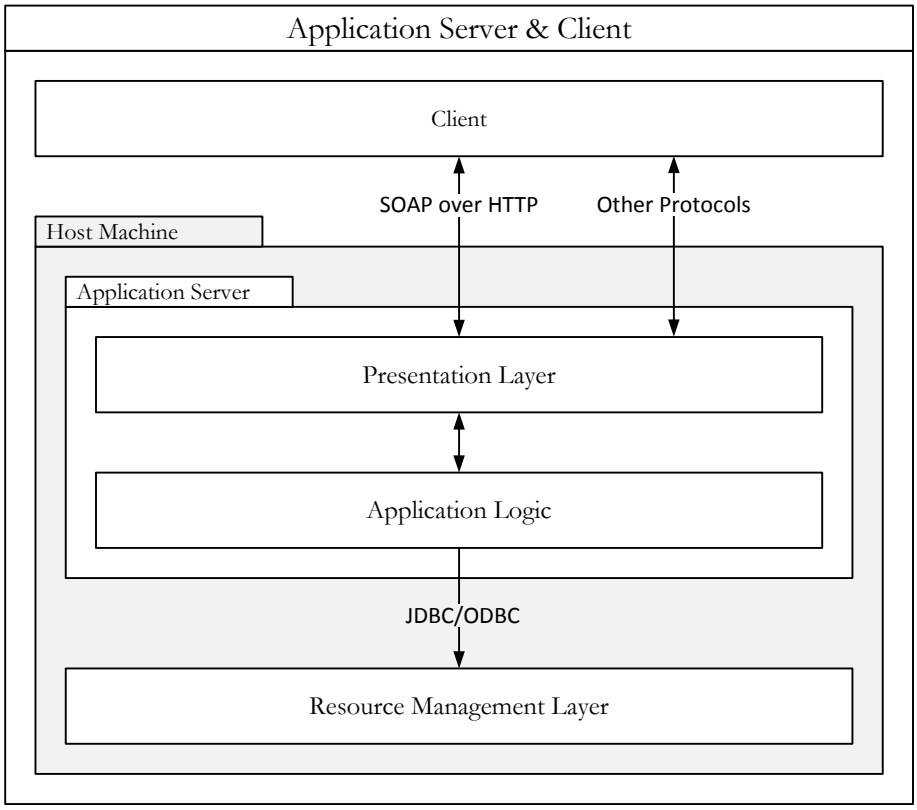

Fig. 2 The architecture of a typical application server with resource management layer

1.2.2 The Discovery Service

The Discovery Service has a role to play in every stage of the Web Service lifecycle. It is used by the Assembler during Design & Development in order to locate information about services which can be used

together to provide new composite services. The Discovery Service can also take on part of the role of TargetManager of a Domain, proving information about previously-deployed services to the Planner during Deployment. If a management infrastructure is in place, the Discovery Service plays an ongoing role in the management control-loop. It provides a source of information for the Planner and, if implemented, may have a managing entity to remove its failed entries and replace them with valid ones.

The Discovery Service is similarly crucial for requestor agent software which dynamically resolves endpoints at run-time: in order for the application to run, the Discovery Service *must* be available and, ideally, populated with correct and up-to-date information. The practicalities of run-time lookup are discussed further in section 2.1 'Managing Change: By Service Consumers' while approaches to improving the reliability, accuracy and performance of the Discovery Service are presented in section 2.2 'Managing Change: By Discovery Services'.

1.2.3 The Service Consumer

Programmers of requestor agent software use a Discovery Service at design time to find information about Web Services and to retrieve WSDL documents. Service interaction software is written using the formats, protocol, and static physical endpoint specified in the Web Service's WSDL description. More sophisticated software may contain only the interfaces of the required service, using UDDI to dynamically resolve endpoints at run-time. When executed by Service Consumers these software applications use the URI encapsulated by a Web Service's WSDL description to contact the application server at the host address and initiate the communication protocol. Encoding the message in document/literal format, the call name, type definitions, and serialized, platform-neutral call parameters are enclosed within a SOAP-formatted XML document and sent over HTTP to the application server. Any return results are received as an HTTP response, parsed, and the return values deserialized back into language-specific objects.

The work presented in this document is largely undertaken for the benefit of Service Consumers: Reliable Discovery Services are important for Service Consumers and reliable Web Service endpoints are vital. Section 2 'Managing Change' identifies the potential problems in Web Service interactions from each actor's point of view and presents the range of approaches applied to make Web Services more reliable, high-performance, and fault tolerant.

1.3 Web Service Composition

In order to implement the interface specified in its WSDL document, a Web Service's business logic may include the invocation of operations offered by other Web Services. A Web Service implemented by

combining the functionality provided by other Web Services is called a 'composite Web Service' and the process of developing composite Web Services is called 'Web Service composition' [4].

Composite Web Services can be written as assemblies or as monolithic implementations. Monolithic composite provider agent implementations directly represent requestor agent software with the exception that the provider agent implementations expose an external interface. The resultant code still suffers from all of the difficulties of writing and maintaining requestor agent software using low-level techniques (converting between XML and scalar variables, locating and accessing Discovery Services, constructing SOAP messages, managing failure, etc.).

Languages for Web Service composition such as XL [41], WSFL [42], and BPML [43] have been developed in an effort to remove low-level technical details from the higher-level process of composition, easing the task of developing and maintaining composite Web Services. These language specifications share many similarities [4] which have converged into an industry standard language for Web Service composition: the Business Process Execution Language for Web Services (BPEL4WS or, more commonly, BPEL) [44].

BPEL documents define and coordinate Web Service invocations using a set of XML constructs called 'activities'. Activities in BPEL can be either 'structured' (used for orchestration) or 'basic' (used for invocation). Basic activities include '*receive*', for receiving a message from a client, '*reply*', for replying to a client message, '*assign*', for assigning data to variables, and '*wait*', for blocking the process for a set period of time. Basic activities can be grouped and executed within structured activities which define their order of execution. Conditional branch logic is provided by a '*switch*' activity while the '*flow*' activity is used for parallel execution of basic activities (or groups of them). A '*while*' activity is provided for looping and '*pick*' can used for event handling.

BPEL documents are executed within a run-time environment typically referred to as a '*composition engine*'. Composition engines receive and reply to invocation requests just like any other Web Service container. On receipt of an invocation request the composition engine creates a 'composition instance' of the composite Web Service which is responsible for executing the activities and calling any necessary external Web Services.

Composite Web Services are advertised and invoked just like any other Web Service: clients interact with composition engines in the same way that they interact with Web Service containers and application servers, using the same messages and messaging protocols as defined in the WSDL. The same service may be

provided as a monolithic implementation and/or as a composite [assembly] implementation and, while their execution environments may differ, both are invoked identically. This basic abstraction is an important feature of Web Service composition as it enables services to be iteratively composed – creating composite Web Services out of other composite Web Services to provide increasingly complex and feature-rich services.

2 MANAGING CHANGE

In contrast to traditional client-server applications, clients in distributed service-oriented environments may communicate with multiple autonomous service endpoints on servers all across the Internet. These Web Service endpoints can become unreachable while their new replacements remain unlisted due to lax requirements on Discovery Services which also may be unreachable. Varying levels of demand for Web Services introduces pressure on Service Providers to implement policies to guarantee service reliability while balancing the provisioning of scare resources. The following sections detail the sources and solutions to managing change from the perspective of each participant in the Web Service lifecycle.

2.1 By Service Consumers

Web Service endpoint descriptions encapsulate a static endpoint which, for any number of reasons, may be temporarily (or permanently) unavailable. In order to improve the probability of successful service invocation, various techniques have been developed which aim to introduce dynamic endpoint discovery, binding, and failure recovery into the execution of requestor agent applications. The ability of a Service Consumer to adapt to changes in Web Service availability depends entirely on how and when binding to these endpoints takes place.

As mentioned previously in section 1.3 ‘Web Service Composition’, client- and server-side requestor agents are identical with respect to binding requirements and both experience the same problems when endpoints become unreachable. With the exception of informal guidance, such as that for developers of Microsoft .Net requestor agents in [45] and [46], there is little work addressing binding techniques in ‘client-side’ requestor agent software. Because a more coherent body of work exists for binding in ‘server-side’ requestor agents, this discussion will be guided by work addressing the ‘how’ and ‘when’ of service binding in Web Service compositions, using the notation and terminology introduced in [47].

A binding of a service into a composition can be defined as the reference used to choose the service to be invoked as a part of the composition [47]. Web Service requestor agent software is written against WSDL documents containing an abstract interface description (`portType`) together with a fixed endpoint (`port`). Developers can bind to a single specific port at design time (static binding) or to a `portType` which can be resolved to a `port` at various points in the future (dynamic binding). Although static binding is the dominant approach used in practice [4] [26] it results in code with tight-couplings between service providers and service requestors and is thus contrary to the central principles of service-orientation.

Dynamic binding is typically performed either at *startup time*, just prior to executing a composition, at *invocation time*, just prior to invoking a service operation, or at *failed invocation time*, in the case of a failed operation or unavailable endpoint. Choosing when to resolve an abstract reference into a concrete endpoint directly affects the likelihood of successful execution of a workflow and can have implications for application performance, particularly when using UDDI for service discovery [12] [14] [15]. In environments where Web Service endpoints are not very reliable, or where their location changes frequently, binding later (i.e. at invocation time) may be more desirable than in reliable environments with fixed endpoint locations. When dynamic binding is significantly expensive, however, binding earlier (i.e. at startup time) can help mitigate the risk of poor execution performance due to a slow Discovery Service [16]. Analyzing the impact of UDDI lookups on service performance, Blake et. al 2007 [16] conclude that when the percentage of services requiring re-resolution was less than 59%, *failed-invocation time* dynamic binding was most efficient when compared with dynamically binding at *startup time*.

Dynamic binding can be incorporated into requestor agent applications by internally modifying the process specification 'in-band', as in Fig. 3, or externally to the process using a proxy as in Fig. 4. Client-side requestor agents typically incorporate failed invocation time dynamic binding in-band, explicitly coding around every service invocation that is intended to be performed in a more robust way [48]. The primary drawback of this approach is the responsibility laid upon Service Consumers to take *all* corrective actions, requiring considerable client-side code that has fixed service-selection logic [49].

The time and effort required to introduce failure recovery techniques directly into executable code can be reduced through the use of extensions to the standard BPEL language. Developers using Self-healing BPEL (SH-BPEL) [50] include annotations in their code which they use to define recovery handlers that will be executed in the event of endpoint errors. SH-BPEL code is pre-processed prior to execution in order to replace annotations with standard BPEL constructs and yields code that can be executed in standard BPEL

engines. While this code could conceivably be directly specified by the designer, the required effort to program the techniques by hand would be considerable [50]. The work described in [51] takes a similar approach but focuses on the application of their endpoint-selection algorithms, using in-band replacement only as a means to apply them at run-time.

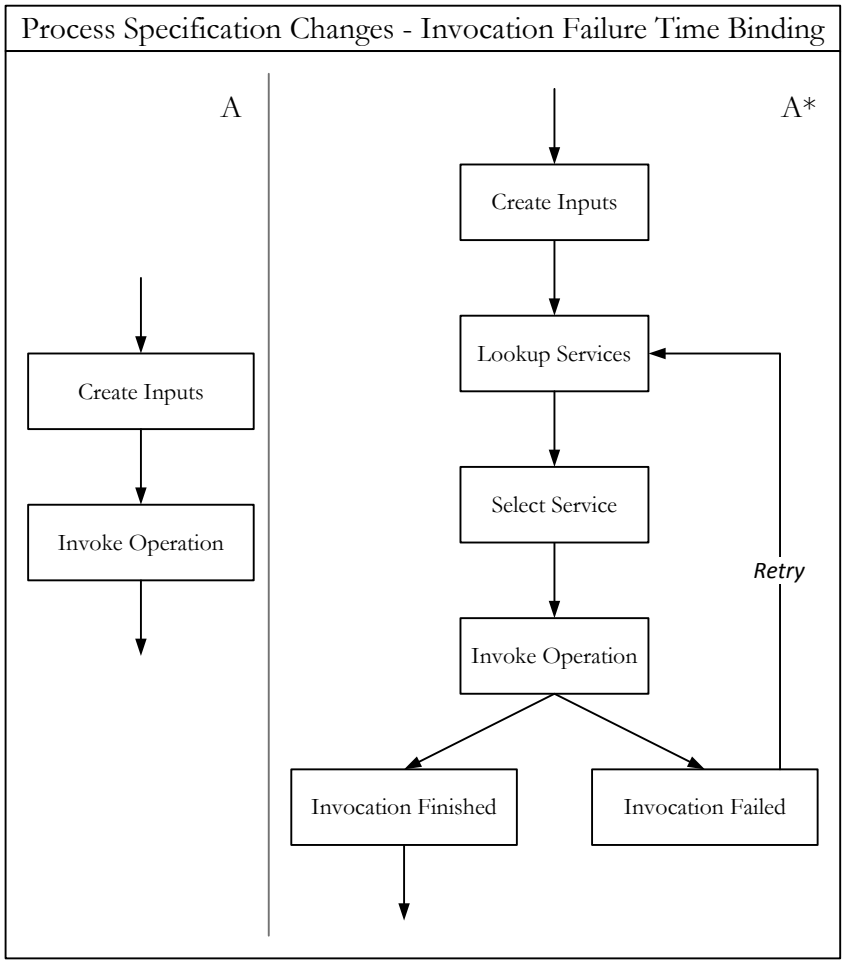

Fig. 3 Dynamically binding 'in-band' alters the structure of process specification 'A' by surrounding the invocation operation with lookup, binding, and failure-detection operations ('A').*

While frequently used by Service Providers for endpoint brokering, the proxy model has a number of benefits when used within the Service Consumer-controlled infrastructure. SH-BPEL, for example, provides a BPEL execution engine plug-in which serves as a proxy in order to introduce dynamic binding at invocation failure time. A similar approach is taken by WS Binder [52] which pre-processes code, statically binding to infrastructure-provided proxies which are created at invocation time for each individual service in the composition. While these approaches do use a proxy they do not meet a central goal of the proxy approach which is to transparently introduce desirable properties – such as dynamic binding and failure

recovery, as in Fig. 4 – into Web Service interactions while imposing limited or no special requirements on developers.

Comparatively transparent extensions to BPEL execution engines are introduced in MASC [49] and Robust Execution Layer (REL) [48] which intercept service invocation requests and apply user-defined policies to the tasks of endpoint selection and failure management. MASC policies are specified in external 'WS-Policy4MASC' documents which are evaluated at startup time, invocation time and, if a language extension is used, at failed invocation time. REL takes a unique, 'hands-off' approach to dynamic binding and does not require any code changes. BPEL documents are written using static bindings to existing endpoints and executed in a BPEL engine with the REL proxy plugin. Service invocation requests and responses are routed and monitored through the plug-in which automatically detects service invocation failures and reacts by locating and redirecting the request to an alternative endpoint. Introduced in the next section, REL relies on a reliable, scalable Discovery Service which includes extra metadata to identify replacement Web Service endpoints which offer equivalent functionality.

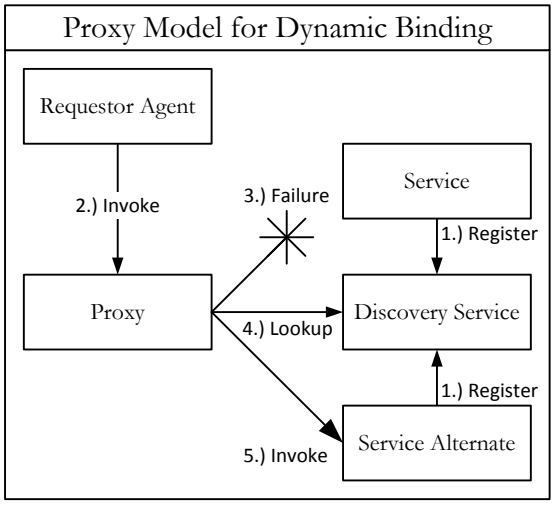

Fig. 4 Detection and rerouting of a failed invocation request using the proxy model.

Whether technology-independent or tightly bound to augmented service code, proxies are realized as distinct functional modules which are separated from the code execution itself, and thus limit the timing of dynamic binding to invocation and invocation failure time. The functional separation allows the proxy to be updated (to change the endpoint selection algorithm or update Discovery Service locations, for example) without interfering with any existing code. Despite this separation of concerns, some proxy approaches do nonetheless take a developer-oriented approach, providing programmers with the means to define how and where dynamic binding and failure recovery should occur while leaving the actual execution of these tasks

to the proxy. The most radical of these approaches is jOpera [47] which enables access to system-level utilities (such as a Discovery Service) directly from the executable code, enabling programmers to write business processes which interact with the execution environment itself, reflectively introspecting and modifying their own document structure in order to control its execution.

The work described in [53] provides an API which allows for fine-grain control over the execution of a business process while abstracting over the details of the Web Service standards involved (such as UDDI). Writing request agent software against the abstract service-oriented tasks of publish/find/bind rather than concrete implementations breaks the tight coupling to specific discovery and communication standards and reduces the amount of code maintenance required when the standards inevitably change. The application of this service-oriented principle into the actual support infrastructure is a powerful concept which is applied throughout the work introduced in this thesis.

Approaches to dynamic binding, whether performed at startup, invocation or failed invocation time and implemented in-band or by proxy, all have a critical reliance upon a Discovery Service. Without a reliable and highly available Discovery Service, dynamic binding will not work and all of the effort applied in order to make requestor agent execution more robust will have been wasted. Appropriately, much work has been applied to the investigation of architectures that can enable the provision of a reliable, scalable and fault-tolerant Discovery Service. The products of this research are presented in the following section.

2.2 By Discovery Services

Discovery Services provide a registry where Service Providers can publish, and Service Consumers can find, information about the existence and location of Web Services and Web Service endpoints. Discovery Services should ideally be scalable, efficient, autonomic, and fault tolerant [54]. While the conceptual view of the Discovery Service role is that of a centralized service broker, as shown in Fig. 5, the architectures of Discovery Service implementations can be variously categorized as centralized, federated, or distributed. Each of these architectural models has tradeoffs of scalability, fault tolerance, performance, and administrative overhead costs and affect the Discovery Service's ability to be reliable and highly available – and thus the Service Consumers ability to successfully complete tasks [55]. The following section details the centralized, federated, and distributed architectural models utilized by Discovery Service implementations and provides a comparison of their strengths and weaknesses.

Discovery Services are most often realized using a centralized architecture [56]. Leveraged as a benefit by SELF-SERV [57], a single server at a well-known location is simple to administer and requires no external replication or coordination with other repositories. As there is only a single point of contact, however, centralized architectures can represent a bottleneck under high load, with centralized UDDI being shown to perform poorly in general [16], and to get worse as the number of entries increases [15]. Crucially, centralized architectures represent a central point of failure for the Discovery Service.

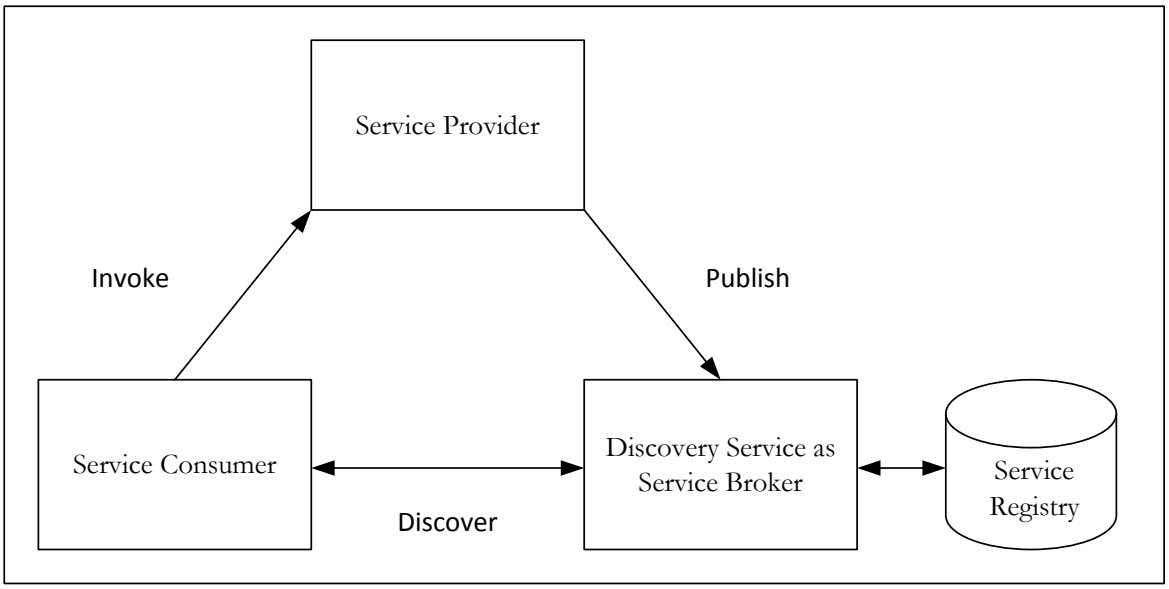

Fig. 5 Conceptual view of the Discovery Service as a centralized Service Broker.

Fault tolerance can be introduced into centralized Discovery Services by replicating registries[2]. While this does provide some resilience to failure, maintaining consistency between replicas requires active coordination and weakens the primary benefit of centralized repositories which is their low administrative overhead. An alternative approach is the use of a federated architecture which gathers together a number of distributed repositories under one conceptually centralized umbrella, with a master peer to maintain consistency and disseminate information among them.

[2] *Hardware-based solutions can also be applied to improve reliability, such as through redundant failovers or 'hot-idling' backup servers, but these are out with the scope and focus of this document.*

Federated architectures, as shown in Fig. 6, represent a hybrid of centralized and distributed architectures: they are comprised of a distributed set of autonomous and inter-communicating registries but are still managed and administered by a centralized entity. While the administration of federated Discovery Services is centralized, the responsibility for storing entries is shared among a set of peers, which spreads the load and resource requirements and provides greater scalability. Federated architectures also provide a degree of fault-tolerance because the failure of an individual registry peer does not affect the entire network.

The METEOR [19] system implements a federated architecture, using a master gateway peer to organize and propagate information amongst different sub-peers. While the entire system remains available even if repositories fail, the entries they hold will be lost. The model can be made more resilient to failure by replicating each repository's entries in other repositories. As replicating entries on arbitrary nodes reduces the independence and autonomy of unrelated registries, the work presented in [58] enables peers to band together and form support 'syndicates'. These syndicates are often grouped based on the content of their registries. These groupings provide the additional benefit of enabling more efficient searches in an otherwise unstructured network.

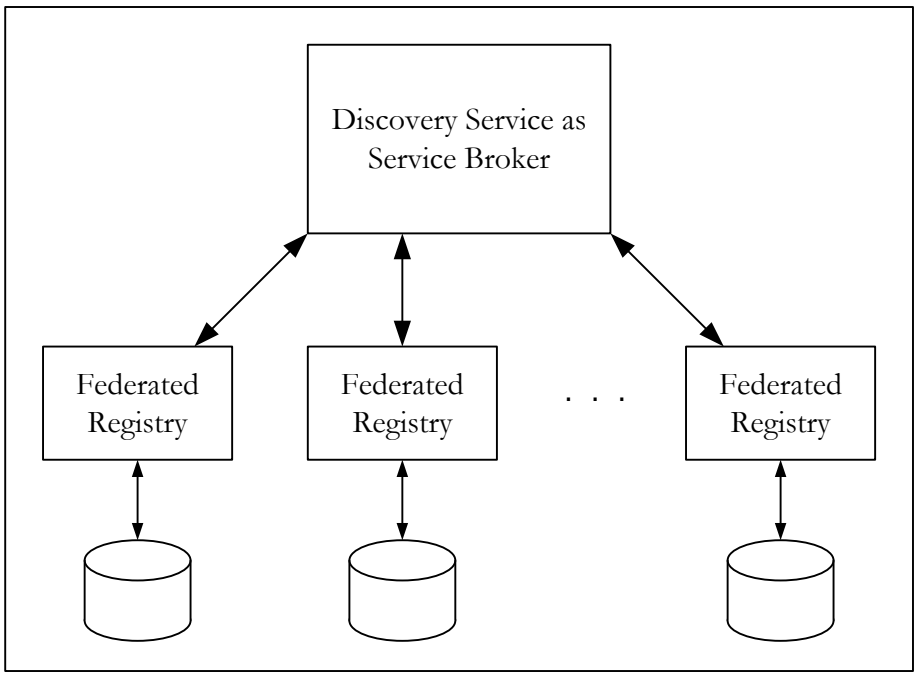

Fig. 6 The federated architecture represents a hybrid of centralized and distributed architectures.

The work presented in [23] utilizes the protocols of Sun's JXTA [59] to enable the construction of ad-hoc and unstructured P2P networks of registry nodes with potentially restricted bandwidth and processing capacity. Peers, or 'common nodes', join the network using JXTA to discover a 'master-host' which assigns them into a

peer group. Each peer group has its own master-host and an undefined number of common nodes which are responsible for monitoring each other's liveness. Master hosts need to be capable of running a 'light' version of UDDI (for storing the entries of the 'common nodes') and are responsible for coordinating peer-group membership and routing lookups. Entries can optionally be replicated between peer groups for failure resilience.

SP2A [54] also uses JXTA to enable the formation of dynamic, ad-hoc registry 'communities', and can operate as either a federated registry (with only 'supernodes' responsible for routing messages) or as a pure distributed registry (where all nodes have equal responsibilities). WSPDS [21] can similarly operate as either a federated or distributed registry and utilizes network-flooding search protocols similar to those of Gnutella [60] and Freenet [61].

The simple client-server interaction seen in centralized approaches is also a feature of federated architectures, as the distributed nature of federated Discovery Services is not visible to Service Consumers. The centralized administration of federated architectures still represents a central point of failure, however, and reduces scalability as it becomes a communications bottleneck under increased load. Pure distributed architectures, as shown in Fig. 7, are completely decentralized and are typically designed with the aim of being both highly scalable and resilient to failure.

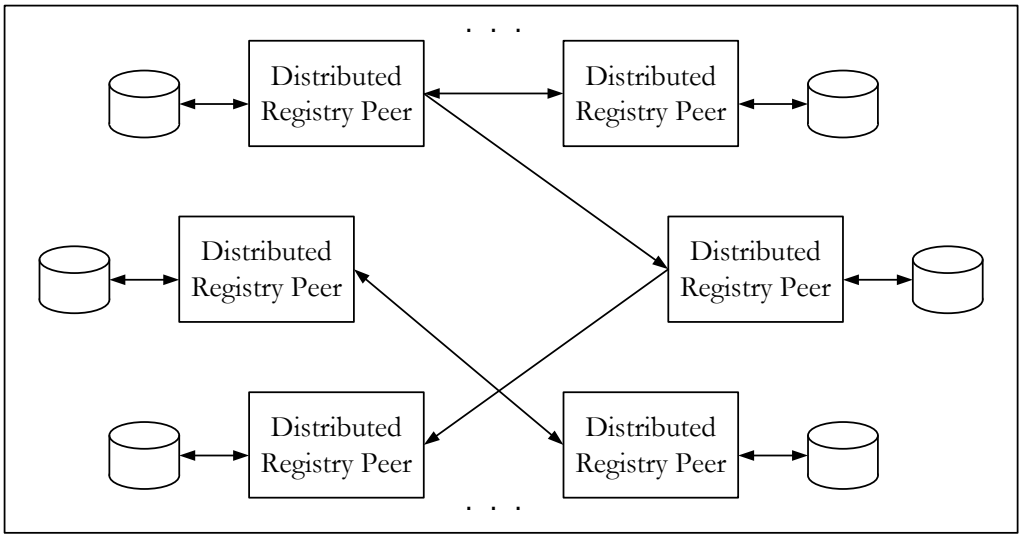

Fig. 7 A distributed network of registry peers.

Fully distributing the responsibility of storing registry entries (and responding to Service Consumer requests) spreads both load and resource requirements, similar to the federated approach. While there is an increased administrative cost associated with maintaining dynamic distributed networks, this too is spread

amongst the participating repository nodes and provides the added benefit of eliminating communications bottlenecks and central points of failure.

Work on peer-to-peer (P2P) overlay networks such as Chord [62], CAN [63], Pastry [64] and Tapestry [65] provide the foundation for the majority of distributed Discovery Service implementations (e.g. [20] [22] [66]). These core technologies provide the protocols for manifesting, maintaining, and routing messages in P2P networks. Registry peers are located using content-addressable routing which introduces a structured address space into an otherwise unstructured network. Because registry peers can join and leave the network at will, replication is often used to provide more resilient mappings (as in [67]).

A primary drawback of distributed Discovery Services is the need for Service Consumers to have specialized knowledge of specific search procedures. To overcome this hurdle, distributed discovery services are often paired with the client-side invocation time dynamic binding techniques detailed previously. Proxy-based approaches in particular provide a high-level abstraction over the low-level details of endpoint resolution, and thus enable different search technologies to be transparently substituted based on the implementation of the chosen Discovery Service. Further, this approach requires no additional effort from Service Consumers who should only notice an improvement in both reliability and lookup performance.

It is also important for Discovery Services to contain accurate and up-to-date information about Web Service endpoints. Unfortunately, most service registries contain entries with broken links to WSDL documents, or WSDL documents that reference unavailable Web Service endpoints [26] with a 2005 survey [68] finding 48% of publicly-available UDDI entries unusable.

A novel method for maintaining up-to-date information is the use of the network topology to inform the removal of inaccurate registry entries. Registry information is kept up-to-date in [23], for example, by relying on the elected leader of each common-node community to send node-failure notifications to their master-hosts, who in turn remove the failed node's entries from the registry.

Instead of defending against registry peer failure, the work presented in [24] embraces it as an indicator of endpoint failure. In this approach, Service Providers themselves can join the peer network and provide the functionality of the Discovery Service, listing only the services they themselves provide. When Service Providers leave the network (whether gracefully or ungracefully) their registry entries leave with them.

This presents a clean method of maintaining accurate and up-to-date registry information without requiring an external management entity.

Discovery Services which aim to be reliable and accurate – regardless of their implementation architecture – are more often operated directly by Service Providers, due to their close knowledge of the status of their domain. The following section recaps the approaches utilized by Service Providers to not only maintain an accurate and up-to-date Discovery Service, but also to ensure that domain resources are applied effectively in order to maintain the performance and availability of Web Service endpoints.

2.3 By Service Providers

Service Providers experience change when endpoints or nodes within their domain fail, and when demand levels rise or fall. Approaches to planning, preparing, monitoring, analyzing and correcting Web Service deployments have been presented in the previous sections which address these lifecycle stages in isolation.

REFERENCES

[1] W3C. (2008). World Wide Web Consortium - Web Standards. [http://www.w3.org/].

[2] W3C. (2007). (27 April 2007). SOAP version 1.2 part 0: Primer (second edition). W3C[http://www.w3.org/TR/2007/REC-soap12-part0-20070427].

[3] W3C. (2007). Web services description language (WSDL). [http://www.w3.org/2002/ws/desc].

[4] G. Alonso, F. Casati, H. Kuno and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Germany: Springer-Verlag Berlin Heidelberg, 2004.

[5] W3C. (2004). Web services architecture. [http://www.w3.org/TR/ws-arch/].

[6] W3C, *Extensible Markup Language (XML) 1.0 W3C Recommendation.* 1998, [http://w3c.org].

[7] Apache Software Foundation. (2008). Apache CXF 2.0 user's guide - WSDL to java. [http://cwiki.apache.org/CXF20DOC/wsdl-to-java.html].

[8] Microsoft. (2008). SPROXY: XML web service proxy generator. [http://msdn.microsoft.com/en-us/library/ztta389h(VS.80).aspx].

[9] Eclipse Foundation. (2008). Eclipse project. [http://www.eclipse.org/projects/].

[10] OASIS. (2008). Universal description discovery and integration (UDDI). [http://uddi.xml.org/].

[11] OASIS. (2008). Organization for the advancement of structured information standards. [http://www.oasis.org]

[12] IBM, H. Adams, D. Gisolfi, J. Snell and R. Varadan. (2004). Best practices for web services. 2008).

[13] G. Saez, A. L. Sliva and M. B. Blake, "Web services-based data management: Evaluating the performance of UDDI registries," in *Proceedings of the IEEE Internal Conference on Web Services,* 2004, pp. 830.

[14] J. Metso, "Suitability of UDDI registry for the web: Performance measurements," University of Helsinki, Finland, Tech. Rep. C-2003-72, 2003.

[15] S. Miles, J. Papay, V. Dialani, M. Luck, K. Decker, T. Payne and L. Moreau, "Personalized grid service discovery," in *Proceedings of the Nineteenth Annual UK Performance Engineering Workshop (UKPEW'03),* 2003, .

[16] M. B. Blake, A. L. Sliva, M. Muehlen and J. V. Nickerson, "Binding now or binding later: The performance of UDDI registries," in *Proceedings of the 40th Annual Hawaii International Conference on System Sciences (HICSS'07),* 2007, pp. 171.

[17] L. Andrade and J. Fiadeiro, "Composition Contracts for Service Interaction," *Journal of Universal Computer Science*.

[18] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsouko and M. Wermelinger, "Coordination for orchestration," in *Proceedings of the 5th International Conference on Coordination Languages and Models,* 2002 .

[19] V. Kunal, S. Kaarthik, S. Amit, P. Abhijit, O. Swapna and M. John, "METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services," *Inf. Technol. and Management,* vol. 6, pp. 17-39, 2005.

[20] L. Quanhao, R. Ruonan and L. Minglu, "DWSDM: A web services discovery mechanism based on a distributed hash table," in *Proceedings of the Fifth International Conference on Grid and Cooperative Computing Workshops,* 2006, .

[21] F. Banaei-Kashani, C. -. Chen and C. Shahbi, "WSPDS: Web services peer-to-peer discovery service," in *Intl. Symposium on Web Services and Applications,* 2004, .

[22] S. Yu, J. Liu and J. Le, "DHT facilitated web service discovery incorporating semantic annotation," in *Lecture Notes in Computer Scienc*eAnonymous Springer Berlin / Heidelberg, 2004, .

[23] J. Lukasz, L. Jaroslaw and D. Schahram, "Web service discovery, replication, and synchronization in ad-hoc networks," in *Proceedings of the First International Conference on Availability, Reliability and Security,* 2006, .

[24] D. Schahram and T. Martin, "Integration of transient Web services into a virtual peer to peer Web service registry," *Distrib. Parallel Databases,* vol. 20, pp. 91-115, 2006.

[25] OASIS. Electronic business using eXtensible markup language. [http://www.ebXML.org/].

[26] S. Hagemann, C. Letz and G. Vossen, "Web Service Discovery - Reality Check 2.0," *International Journal of Web Service Practices,* vol. 3, pp. 41-46, 2008.

[27] Web Services Interoperability Organization. (2008). About WS-I - overview. [http://www.ws-i.org/about/].

[28] Ballinger,K., D. Ehnebuske, M. Gudgin, M. Nottingham and P. Yendluri. (2004). (April 16 2004). WS-I basic profile version 1.0. Web Services Interoperability Organization [http://www.ws-i.org/Profiles/BasicProfile-1.0-2004-04-16.html].

[29] UDDI.org, *UDDI Version 2.04 API, Published Specification*. 2002. [http://www.uddi.org].

[30] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte and D. Winer, "Simple object access protocol (SOAP) 1.1," W3C, 8 May 2000. 2000.

[31] International Organization for Standardization, "ISO/IEC 29361:2008 Information technology - Web Services Interoperability - WS-I Basic Profile Version 1.1," June 30th, 2008, 2008.

[32] A. Thomas, "Enterprise JavaBeans technology; server component model for the java platform," Sun Microsystems Inc., 1998.

[33] Apache Software Foundation. (2004). Apache axis. [http://ws.apache.org/axis/].

[34] Sun Microsystems. (1998). Technical Report (May 30, 1998). JDBC overview. [http://java.sun.com/products/jdbc/].

[35] Microsoft Corporation, Microsoft Corporation. (2008, Microsoft open database connectivity (ODBC). [http://msdn.microsoft.com/en-us/library/ms710252.aspx].

[36] G. Alonso, F. Casati, H. Kuno and V. Machiraju, "Application servers," in *Web Services: Concepts, Architectures and Applications*Anonymous Heidelberg, Germany: Springer-Verlag Berlin Heidelberg, 2004, pp. 102.

[37] Sun Microsystems, (2002). Sun one application framework. 2008.

[38] IBM. (2008). IBM - WebSphere application server.

[39] Microsoft Corporation. (2008). Microsoft .NET framework.

[40] Oracle Corporation. (2008). BEA WebLogic server. [http://www.bea.com/content/products/weblogic/server/].

[41] D. Florescu, A. Grunhagen and D. Kossmann, "XL: An XML programming language for web service specification and composition," in *Proceedings of the 11th International World Wide Web Computer Conference (WWW02)*, Honolulu, Hawaii, USA, 2002.

[42] F. Leymann, "Web services flow language. version 1.0." International Business Machines Corporation (IBM), May 2001. 2001.

[43] A. Arkin, "Business process modeling language 1.0. technical report," BPMI Consortium, June, 2002. 2002.

[44] IBM, BEA Systems, Microsoft, SAP AG and Siebel Systems. (2003). (2007). BPEL4WS specification. [http://www.ibm.com/developerworks/library/specification/ws-bpel/].

[45] Januszewski,K. and Microsoft Corporation. (2007). Using UDDI at run time, part I. [http://msdn.microsoft.com/en-us/library/ms953944.aspx].

[46] Januszewski,K. and Microsoft Corporation. (2007). Using UDDI at run time, part II. [http://msdn.microsoft.com/en-us/library/ms953948.aspx].

[47] C. Pautasso and G. Alonso, "Flexible binding for reusable composition of web services," in *Software Composition*Anonymous Springer Berlin / Heidelberg, 2005, pp. 151-166.

[48] T. Friese, J. Müller and B. Freisleben, "Self-healing execution of business processes based on a peer-to-peer service architecture," in 18th Int. Conference on Architecture of Computing Systems, Innsbruck, Austria, 2005, .

[49] A. Erradi and P. Maheshwari, "Dynamic binding framework for adaptive web services," in *Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, 2008, pp. 162-167.

[50] M. Stefano, M. Enrico and B. Pernici, "SH-BPEL: A self-healing plug-in for ws-BPEL engines," in *Proceedings of the 1st Workshop on Middleware for Service Oriented Computing (MW4SOC 2006)*, Melbourne, Australia, 2006, .

[51] L. Steffen, A. Anupriya, S. Rudi and G. Stephan, "Preference-based selection of highly configurable web services," in *Proceedings of the 16th International Conference on World Wide Web, Banff*, Alberta, Canada, 2007, .

[52] M. D. Penta, R. Esposito, M. L. Villani, R. Codato, M. Colombo and E. D. Nitto, "WS binder: A framework to enable dynamic binding of composite web services," in SOSE '06: *Proceedings of the 2006 International Workshop on Service-Oriented Software Engineering*, Shanghai, China, 2006, pp. 74-80.

[53] A. Michlmayr, F. Rosenberg, C. Platzer, M. Treiber and S. Dustdar, "Towards recovering the broken SOA triangle: A software engineering perspective," *in IW-SOSWE '07: 2nd International Workshop on Service Oriented Software Engineering*, Dubrovnik, Croatia, 2007, pp. 22-28.

[54] M. Amoretti, F. Zanichelli and G. Conte, "SP2A: A service-oriented framework for P2P-based grids," in *Proceedings of the 3rd International Workshop on Middleware for Grid Computing*, Grenoble, France, 2005, .

[55] S. Dustdar and M. Treiber, "A View Based Analysis on Web Service Registries," Distrib.Parallel Databases, vol. 18, pp. 147-171, 2005.

[56] S. Dustdar and M. Treiber, "WiZNet - Integration of different Web service Registries," .

[57] B. Benatallah, Q. Z. Sheng and M. Dumas, "The Self-Serv environment for Web services composition," *IEEE Internet Computing*, vol. 7, pp. 40-48, Jan-Feb, 2003.

[58] M. P. Papazoglou, B. J. Krämer and J. Yang, "Leveraging web-services and peer-to-peer networks," in *Proceedings of the 15th International Conference on Advanced Information Systems Engineering (CAiSE 2003)*, 2003, pp. 485-501.

[59] L. Gong, "Project JXTA: A technology overview," Sun Microsystems Inc., 2002.

[60] G. Kan, "Chapter 8: Gnutella," in *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*, A. Oram, Ed. O'Reilly, 2001.

[61] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong, "Freenet: A distributed anonymous information storage and retrieval system," in *Designing Privacy Enhancing Technologies: Lecture Notes in Computer Science 2009*, H. Federrath, Ed. Springer, 2000, pp. 46-66.

[62] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan, "Chord: A scalable peer-to- peer lookup service for internet applications," in *ACM SIGCOMM 2001*, San Diego, CA, USA, 2001, pp. 149-160.

[63] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A scalable content- addressable network," in *SIGCOMM'01*, San Diego, CA, USA, 2001, pp. 161-172.

[64] A. I. T. Rowstron and P. Druschel, "Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, Heidelberg, Germany, 2001, pp. 329-350.

[65] B. Y. Zhao, J. Kubiatowicz and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," University of California at Berkeley, 2001.

[66] D. Talia, P. Trunfio, R. Orlando and C. Silvestri, "A DHT-based peer-to-peer framework for resource discovery in grids," *in Achievements in European Research on Grid Systems* Anonymous Springer US, 2008, pp. 123-137.

[67] S. J. Norcross, A. Dearle, G. N. C. Kirby and S. M. Walker, "A peer-to-peer infrastructure for resilient web services," in *IEEE International Workshop on Advanced Architectures and Algorithms for Internet Delivery and Applications (AAA-IDEA 2005)*, Orlando, Florida, USA, 2005, pp. 65-72.

[68] D. Schahram and S. Wolfgang, "A survey on web services composition," *Int. J. Web Grid Serv.*, vol. 1, pp. 1-30, 2005.