Active UDDI - an Extension to UDDI for Dynamic and Fault-Tolerant Service Invocation

Mario Jeckle, Barbara Zengler
DaimlerChrysler Research and Technology
Research Information and Communication / Data and Product Management
Ulm, Germany
email: {mario.jeckle, barbara.zengler}@daimlerchrysler.com

Abstract. UDDI, Universal Description, Discovery, and Integration, represents a directory for the publication and querying of categorized Web services. Publication and query are performed by utilizing UDDI's Application programming interface (API), which employs SOAP as a communication instrument.
By offering an invocation API in addition to two other types for examination, UDDI allows clients to search for and subsequently invoke specific Web services. Failures in invoking already sought and, on the application side, statically cached Web services typically result in re-querying the registry.

However, an application's reaction time in response to changes is limited due to UDDI's replication latency, i.e. the amount of time it takes for changes to entries stored inside the UDDI repository to be propagated to all UDDI nodes.

This paper proposes a mechanism termed *active UDDI*, which allows the extension of UDDI's invocation API in order to enable fault-tolerant and dynamic service invocation.

1 Introduction

UDDI represents a directory for the publication and discovery and location of categorized Web services. The UDDI specifications [IIea01] published by an industry initiative[1] introduce *data structures*, *API specifications* as well as a directory-specific *replication process* and an *operator's specification*. The abstract notion of a *UDDI directory* is implemented by an application termed *UDDI registry*, which is comprised of several data-holding entities called the *UDDI nodes*. All of these nodes which additionally allow publication of service-

[1] including Microsoft, SAP, Intel, HP, Compaq, SUN, and Verisign

related information are further rubricated *UDDI operator nodes*.

UDDI offers a set of application programming interfaces (APIs), which can be used to publish or search information stored within the directory.
This API defines interfaces utilizing W3C's *XML protocol/SOAP*[GHM+02][2] as a communication instrument. Therefore, the UDDI directory as a whole can be regarded as a publicly available Web service.

Clients desiring to publish or retrieve information send SOAP messages to one of the nodes. After processing, the node sends back the result to the client as a SOAP message. If the processing results in an error, the node uses SOAP to send back a fault message indicating the type of error it encountered.

Queries, search results, and error reports directed to and emitted by the UDDI repository are formulated in a language conforming to W3C's *Extensible Markup Language* (XML) [BPSMM00]. The definition for this language and the UDDI data structures are given in several XML schema files.

The *UDDI API* provides interfaces for three patterns of request: the *browse pattern*, the *drill-down pattern*, and the *inquiry pattern*.
Shared semantics of these patterns is that they solely enable different ways of accessing the data stored inside the nodes of a repository but do not permit restriction of the services queried by other characteristics as defined by the service provider when initially publishing the service to a UDDI node.

Though providing basic support for remote service invocation, UDDI does not support dynamic service invocation within a network of distributed services. Networks based on Internet technology are subject to diverse changes resulting from the network protocol design. Thus, applications using Internet technologies to fulfill their functionality need to be capable of reacting to incessant and immediate changes such as network partitioning and, as a result, temporary unavailability of parts of the network where services reside. For this reason, the network itself virtually stores information about the dynamic state of a service. A repository capable of dynamic reaction to errors should be able to detect changes in the availability of a utilized service or at least react to it. Additionally, the repository should be capable of transparently replacing the - most likely only temporarily - unavailable service by an alternative one that fulfills the same task.
This behavior is referred to as *dynamic service invocation* within this paper.

With dynamic service invocation only one aspect of fault tolerance, UDDI does not at all support handling mechanisms for fault situations commonly encountered in distributed systems[3]. Applications, however, should be supported

[2]Originally SOAP served as abbreviation for *Simple Object Access Protocol*, but since W3C took on responsibility for the standardization this is no longer the case.

[3]such as a decrease in quality of service which could be caused by an increase in response

by the repository in coping with such situations in order to maintain a uniform level of quality of service.

Active UDDI as proposed in this paper is an extension to UDDI, which enables applications to dynamically react to fault situations as outlined in this paper using the example of network error situations.

1.1 The UDDI Data Model

For a basic understanding of how UDDI works, a short overview of the data structures utilized in registering information is provided in the following.

UDDI distinguishes five core data structures: *businessEntity*, *businessService*, *bindingTemplate*, *tModel*[4] and *publisherAssertion*.
BusinessEntity embodies the known information about a registered body[5], including descriptive information such as the body's name, contact information for human contacts, both the publisher's name and the custodian's name as well as descriptive and technical information about the *businessServices* it offers.
It is possible to supplement additional body identification information such as tax identifiers to the *businessEntity* or categorizing information such as standardized industry, product or geographical codes. Also, these standardized codes may be validated by UDDI operator nodes.

The *businessService* structure is contained in a businessServices structure and aggregates descriptive information about a specific Web service such as the service's name, description, and its *bindingTemplate*. As with the *businessEntity*, additional categorizing information such as standardized industry, product or geographical codes may be appended. *businessService*s may be utilized as a projection, i.e. an already published service can be reused or shared within several *businessEntity* elements by including the service reference in the *businessEntity*.

*bindingTemplate*s contain information for defining the technical entry point for a specific Web service and describing service-specific technical characteristics including parameter- specific settings. For instance, the information contained may consist of a descriptive text, the service parent's key which relates the bindingTemplate to its providing businessService and either an indication for the service access location or a reference to a different bindingTemplate describing the service in more detail[6]. Additionally, a technical fingerprint[7].

---

time or repeated erroneous data transfers.

[4] technicalModel

[5] This could be an organization, a company or any other kind of service-providing entity.

[6] This allows the remote hosting of services or a grouping of service descriptions

[7] This technical fingerprint is represented by the combination of the tModel references contained in the bindingTemplate within a *tModelInstanceDetails* element

These sets of tModel references can be searched for by an interested party. The search result includes solely those bindingTemplates that match the specific fingerprint. These fingerprints may additionally be registered within the UDDI registry of a service and can be contained within a *bindingTemplate*.

tModel data structures represent abstract structures for the description of data. Consisting of a unique key, a name, and an included and/or external description, they allow not only the specification of concepts but also a technical description of Web services. Therefore, the UDDI specification defines two application domains for tModels, namely compatibility checks utilizing technical fingerprints and namespace references. With the key of a tModel representing its technical fingerprint or namespace reference, a publisher is able to specify compliance to a concept by including a reference to the specific tModel in an application. Consequently, client applications may then search for Web services compatible to the specific concept[8].

A *publisherAssertion* structure allows the declaration of relationships between businessEntity structures. Large organizations may publish several businessEntities which represent their subordinate companies. The validity of business relationships has to be agreed on by both the parties involved: both have to mutually define the relationship. A publisherAssertion consists of the keys of the related bodies and a reference to the tModel representing the relationship.

Every UDDI data structure contains its own unique identifier in terms of a UUID as specified in [ISO96]. The three data structures *businessEntity*, *businessService*, and *bindingTemplate* are related to each other via a containment relationship. A *businessEntity* may contain one or more *businessService*s, which again may contain one or more *bindingTemplate*s.

UDDI distinguishes the relations *containment* and *reference*. Whereas a containment implies that a specific data instance may be contained by only one parent at a specific point in time, references may occur in several places simultaneously. References are typically found in lists which are not individual instances themselves. Any key values directly contained in structures that are not one of the five core structure types themselves are references. For example, the *bindingTemplate* references *tModel* data structures, thus allowing a specific tModel instance to be included in several bindingTemplates at a specific point in time.

1.2 The UDDI Replication Process

Publicly available UDDI nodes, similar to the DNS system, together form a service that, while appearing to be virtually a single component, is composed of an arbitrary number of operator nodes. They are called the *UDDI cloud*. An

[8] Applications must be aware of the relation between the abstract concept and its representing tModelKey in order to utilize this mechanism.

operator node is responsible for the data published at this node: in UDDI terms, it is the custodian of that part of the data.

UDDI data must be kept consistent between the nodes of a UDDI cloud. This goal is achieved by the application of *UDDI's data replication process*[9] within the participating nodes. Furthermore, the UDDI replication process is the only available mechanism allowing cloud- participating nodes to communicate data changes. The process utilizes XML and SOAP for inter-node communication. With regard to the development of applications using UDDI, the possibilities of an application implementing the functionality of a whole UDDI registry or parts of it[10] as well as an application solely acting as a UDDI client must be equally considered.

For replication purposes, *change records*[11] consisting of the originating node's UUID, its *update sequence number*[12] and the data payload are transferred between UDDI nodes with each operator node recording the transferred and processed messages.

Each node's replication state is represented by a *highWaterMarkVector* containing the node's identity in terms of an *operatorNodeID* and information about the latest change in terms of an update sequence number. The highWaterMarkVector consists of the same semantic information as the ChangeID in the change record. The UDDI specification does not mention why there are two structures which carry the same information.
Replication is configured in a cloud-wide master configuration file in XML format containing a serial number[13], the time of the last update, contact information, and replication time configuration. Replication time is defined in two intervals in the unit *hour*:

- The interval defining the assumed time of change visibility at all nodes.
- The interval defining the maximum time an individual is to wait before utilizing change requests.

Additionally, the configuration file aggregates information about participating operator nodes including each node's target replication URL *soapReplicationURL* indicating a secure HTTP connection. Therefore, nodes need to be able to act as TLS1.0 servers and clients[14] with at least RC2 and RC4 40-bit encryption

[9]The data replication process is specified within the *UDDI Replication Specification* [IIea01].

[10]i.e. participating within the cloud in the role of an operator node

[11]A change record structure is transmitted between nodes to indicate the kind of change that is to be processed together with the data to be changed.

[12]An increasing sequence number which is generated locally at every node, indicating the most recent change

[13]Similar to the serial number of DNS SOA records, the number is incremented with every update.

[14]Each operator must therefore acquire an X.501 certificate.

and MD5 message authentication algorithms. Replication process message flow is driven by a global communication graph specified within the configuration file. The graph configuration element lists both the participating nodes by their IDs and the message types that are to be controlled. Fallback edges may be installed.

UDDI v2 offers a mechanism to announce the availability of changes utilizing the *notify_changeRecordsAvailable* element. A node employing this mechanism informs other nodes about the availability of changes, providing its ID together with its own replication state and other nodes' states known within the *notify_changeRecordsAvailable* message. Application of this notification instrument can reduce the retrieval time the polling mechanism would take. Nodes receiving such a message are subsequently required to ask for changes within the time interval specified in the configuration file. Implementation of this notification mechanism is purely optional.
This mechanism is an approach toward solving our problem; yet since nodes are required to ask for changes within an interval that can only be defined in the unit hour, it is not applicable for highly dynamic change information, e.g. when a network temporarily partitions.
Replication is initiated at the UDDI nodes by the caller, i.e. the receiving node, submitting a `get_changeRecords` request message. Within this message type, nodes provide their current highWaterMarkVectors to communicate their state of replication to other participating nodes. In addition, nodes may try to limit the number of messages they wish to receive.
To accommodate bug detection and processing, the UDDI Replication Specification offers a process that helps in detecting, dealing with, and managing the following replication errors:

- Invalid record validation: A receiving node's UDDI implementation cannot validate the incoming change request due to erroneous validation implementation (such as overly aggressive checking).
- Invalid interim representation: An intermediary node inaccurately handles the change record.
- Invalid generation: The originating node poorly generates the change record.

2 Active UDDI

2.1 Basic Idea and Background

The several UDDI processes and mechanisms discussed in the above cover solely operational aspects of the UDDI cloud, data management, and replication aspects. They are designed and suitable for dealing with explicitly published changes to the registry data, which are typically done by operators or publish-

ers[15].
While these processes can be regarded as an approach to automatically handle changes in the registry, they do not represent a solution for the problem of dynamic service invocation or fault tolerance. This is illustrated in more detail in the following paragraphs.

Management of changes to information stored in one of the UDDI nodes participating can be slightly automated by utilizing the replication processes presented above within a UDDI registry.
Controlled by the *global* configuration file, the registry nodes replicate regularly, thus automatically propagating changes within the registry. This data replication is a process of long duration: the replication configuration allows specification of replication time intervals in hours. This time-consuming endeavor makes it poorly suited for real-time dynamic service invocation.

However, UDDI replication covers exclusively those changes that were explicitly published at the registry. Types of changes covered are the registration of new data, changes in the information registered within the data structures, data deletion, and changes in custody for a given datum.
All these changes necessitate publisher intervention since they must be explicitly declared by a UDDI operator or publisher. This could be done automatically by the use of automated facilities for submitting change requests on the publisher or operator sides. Within the UDDI process, publishers and operators are merely capable of changing data they originally published, i.e. data they have the custody for. For example, if the change in availability for a service, i.e. the service has become temporarily unavailable, is to be reported to a UDDI registry, this cannot be done by a party which detects the unavailability, even if this is a trusted party.

Yet the realization of a dynamic infrastructure for applications demands just these aspects. The need for dynamic reaction to immediate changes such as the temporary unavailability of a service must be met, especially in a network environment where Web services are likely to be reside. Applications relying on the functionality of distributed Web services must be able to collaborate dynamically. Therefore, pivotal aspects such as fault tolerance and dynamic reaction must be considered in addition to the basic replication mechanism as provided by UDDI. Applications must be supported in implementing aspects of fault tolerance. Trusted parties[16] must be able to report errors. These errors should be recorded by the registry and taken into consideration when information about an erroneous service is provided.

[15]Operators can allow third parties to publish UDDI data. Therefore, these publishers are explicitly registered by the UDDI operators. They may also be assigned a limited space of unique keys for information registration purposes.

[16]The term trusted parties here explicitly includes Web services. Trust is provided either by authentication means or in terms of a distributed consensus.

2.2 Active UDDI as an Extension to UDDI

For the further discussion of active UDDI, it is assumed that the Web services participating in the service network are interwoven and interdependently utilize each other.

Active UDDI's basic approach is an extension of the existing UDDI infrastructure without requiring changes to the data structures or the APIs presented above. Since a UDDI registry represents itself to the outside as an invocable Web service, active UDDI merely adds an additional Web service, subsequently referred to as the active service, to a UDDI registry which serves as a single entrance point for both the inquiry and publishing APIs within the participating network of Web services, the clients. The extension approach leaves the existing UDDI registry untouched: it is not necessary to change existing data structures. Additionally, no changes need be made to the UDDI APIs as additional interfaces allowing services to use its functionality are provided instead. Thus, the active UDDI extension can be applied to any existing UDDI registry.

Active UDDI's functionality is as follows:
The *active service* locally mirrors the UDDI registry's data pool and implements the UDDI API calls. It can therefore be described as a proxy located between the clients and the UDDI registry. While it participates in the UDDI replication process, it does not have custody of any portion of registration data. However, it implements and offers the additional features necessary to allow real-time dynamic service invocation and fault tolerance.

Mirroring the UDDI registry data, the active service maintains a list of available Web services, which is a subset of the services stored within the UDDI registry. Also, the active service is able to process unavailability (or availability as the case may be) messages originating from its clients indicating that a service is temporarily unavailable. Hence, active UDDI introduces state information about the various Web services. Through continuous maintenance of this state information, the active service is able to provide a statement about the current availability of a service.

Upon reception of an unavailability message, the active service blinds out the affected Web service from its list of available Web services: it is temporarily not included in the node's responses to the inquiry API. When the service becomes available again, it is faded in and thus included in the node's responses.

The temporary changes which are recognized by the active service do not propagate throughout the UDDI cloud. The active service locally registers these changes and, in its role as proxy, prevents the affected services from being reported in response to a client's queries. The original data which is held at the UDDI cloud is not impacted by the active service's operation. The active service does, however, accept change queries by clients but forwards them to the node

in custody of the specific data portion.

As mentioned before, unavailability and re-availability are indicated by the participating clients by transmitting an appropriate SOAP message to the active service. However, not just any arbitrary Web service is permitted to announce a change in the state of a particular Web service. An active UDDI operator has to set up a policy that specifies when announcements are to be accepted. And this policy needs to consider authentication. An active service may allow specific, trustworthy clients to report changes in the state of another client. Therefore, the reporting client may authenticate itself to the server by the use of a digital certificate which clearly reveals its identity.

Additionally, the active service may require its clients to perform a distributed consensus about the change in state of a specific client. The application of a policy for a distributed consensus solves the problem of network partitioning. While a Web service may be unavailable for a particular client service due to network problems at the client's side, it might well remain available for other client services. Marking the service as unavailable merely because a particular client reports it to be unavailable would be intolerable for some application scenarios.

If a service is marked as erroneous, the active service is able to determine alternate services which offer the same functionality. Services are grouped according to their functionalities by comparing their WSDL descriptions. Thus, the active service can offer an alternative , i.e. a service with the same functionality belonging to the same group, which the client may invoke in order to fulfill its task. Furthermore, the active service allows its clients to participate in a publish-subscribe mechanism if desiring to be informed of changes affecting a Web service or of a category of Web services they wish to monitor. Clients can inform the active service about Web services or categories they wish to track. The active service keeps a list of clients and the changes to be tracked. When a listed service or category is affected by either a temporary or permanent change[17] the active service sends the clients an update message indicating the change that occurred.

3 Conclusion

The concept of active UDDI proposed in this paper implements a proxy that is able to actively react to changes and thus supports applications which make use of distributed Web services. It is an approach geared to meeting the requirements of fault tolerance in distributed systems based on Web service technology and is illustrated using the example of how temporary service unavailability is handled. Additionally, the concept offers implementation of a publish-subscribe

[17] The active service detects this either by receiving a message from its clients or from changes within the replication process

mechanism. The proposed UDDI extension can be applied to any UDDI implementation by adding an additional Web service that implements its functionality to the existing infrastructure. The benefit is that this infrastructure does not need to be changed.

References

[BPSMM00] T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler, editors. *Extensible Markup Language (XML) 1.0 (Second Edition)*. World Wide Web Consortium, W3C, October 2000. http://www.w3.org/TR/2000/REC-xml-20001006/.

[GHM+02] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, and H. F. Nielsen, editors. *SOAP Version 1.2 W3C Last Call Working Draft*. World Wide Web Consortium, W3C, June 2002. http://www.w3.org/TR/2002/WD-soap12-part1-20020626/, http://www.w3.org/TR/2002/WD-soap12-part2-20020626/.

[IIea01] IBM, Intel, and Microsoft et al., editors. *UDDI Version 2.0 Specifications*. UDDI.org, June 2001. http://www.uddi.org/pubs/DataStructure-V2.00-Open-20010608.pdf, http://www.uddi.org/pubs/Operators-V2.00-Open-20010608.pdf, http://www.uddi.org/pubs/Replication-V2.00-Open-20010608.pdf, http://www.uddi.org/pubs/ProgrammersAPI-V2.00-Open-20010608.pdf.

[ISO96] ISO, editor. *ISO/IEC 11578: Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)*. ISO, Geneva, CH, 1996.