DeW: A Dependable Web Services Framework[1]

Esam Alwagait and Shahram Ghandeharizadeh
Department of Computer Science
University of Southern California
Los Angeles, CA 90089, USA
alwagait@usc.edu, shahram@usc.edu

Abstract

Web Services (WSs) correspond to conceptual entities with well defined interfaces published by different organizations. For example, with businesses, a WS might correspond to a business process to be invoked by other WSs and Internet applications. To increase availability of a WS, an organization might replicate it across different nodes. This study focuses on data intensive applications that (a) expose a conceptual entity as a Web Service (WS) and (b) disperse copies of their WSs across the nodes of a distributed environment to enhance both performance and availability. We describe the design and implementation of a Dependable Web services (DeW) framework to realize physical-location-independence.

Physical-location-independence means a plan will execute as long as a copy of its referenced WSs is available. This concept enables the client proxy objects to continue operation in the presence of both failures and WS migrations that balance system load.

1. Introduction

Web Services (WSs) advocate a "software and data as a service" paradigm. They are envisioned as the building components of cooperative applications that communicate using a network, i.e., Internet, an intranet, or both. With data intensive applications, a conceptual object or a business process might be implemented as a WS. These WSs might be dispersed across a collection of nodes. For example, with a document management system, a document might be implemented as a WS that embodies data, text, figures, animations, simulation models, observations, etc. Different documents might be assigned to different nodes available in a computing environment in support of inter-request parallelism, i.e., independent execution of requests referencing different WSs. Popular documents might be replicated multiple times on different nodes in order to prevent formation of hotspots and bottlenecks.

A final vision of WSs is an environment for dynamic composition of WSs in support of a user query. Several frameworks strive to realize this vision, e.g., Global XML Web Service Architecture (GXA) [21], the Business Process Execution Language (BPEL4WS) [10], XL [8, 7]. A key challenge is to execute plans in a physical-location-independent manner. This means the plan will execute as long as a copy of the referenced WS is available. This criterion is important because it frees the end user (and programmers) by requiring the system to resolve the location of a WS in the presence of both (a) WS migrations to balance load, and (b) node failures that render a copy of a WS unavailable.

Physical-location-independence is essential for a "what-oriented" interface that enables users to focus on their target application instead of the location of a WS. To illustrate, consider a neuro-scientific document repository that represents a document as a WS. It might provide methods that output traditional data types such as text, images, animations, video clips, etc. In addition, this WS might expose the raw data sets that are the basis of its observations, its processed data sets (e.g., produced using linear regression), simulation models that consume raw data as context to describe and explain a certain behavior, etc. This repository may consist of many WSs organized in a taxonomy for quick browsing and retrieval. A scientist at a remote node may implement an experimental model using the simulation models available from one or more of these WSs(published documents). Now, assume that two WSs that reside on a single node are used extensively and one is moved to a less utilized node. Without

[1] This research was supported in part by an unrestricted cash gift from Microsoft research.

special precautions, the scientist's experimental model might observe failures because its proxy is no longer referencing a valid remote object. With a "what-oriented" interface, the proxy object resolves this change in the environment without disrupting the activities of the scientist.

Physical location independence is also appropriate for performance oriented middle-wares that dynamically adjust the placement of WSs on a collection of nodes to enhance performance, e.g., Ribbet [30].

As an example, consider an airline information management system that exposes each flight as a WS on the network. A Flight WS encapsulates data such as the departure and destination city of that flight, the current number of available seats, the flight's current status (sitting at an airport, mid-flight, arrived), etc. New Flight WSs are created and registered with brokers on the network. These brokers enable clients to discover Flight WSs, based on some query criteria, such as the departure and destination cities, scheduled departure times, etc. When a client locates a suitable Flight WS, it caches its proxy to invoke methods on the remote Flight WS, i.e., many clients may cache the proxy of a WS.

The concept of WSs along with the distributed nature of airline reservation application enhances system performance. To illustrate, assume 90% of system workload consists of queries with the remaining 10% performing updates. If a centralized transaction processing system is connected via a 40 Megabit per second (Mbps) connection to the Internet, and each query/transaction exchanges 1,000 bits with this server, its network connection can support at most 42,000 requests per second.

Consider an environment where WSs are scattered across many nodes. Each transaction that updates a Flight WS must also be directed to the centralized transaction processing system. However, a WS may respond to queries without contacting the centralized server. (Note that we are distributing, and not replicating WSs.) Assume that each node hosting a WS is connected via a 1 Mbps connection to the Internet. Each node may perform 945 operations: 105 transactions and 840 queries[2]. This is because each transaction now requires 2,000 bits: 1,000 bit from the client to the node containing the relevant WS plus another 1,000 bits from this node to the centralized transaction processing system. With the centralized server performing 42,000 transactions per second, this environment can support a workload of 378,000 requests per second before the network bandwidth of the centralized server becomes fully utilized. This factor of 8 improvements in throughput requires a total of 400 nodes along with an even distribution of the workload across them.

A challenge of a middleware such as Ribbet is to enable clients to observe uninterrupted service when a WS migrates from one node to another. This challenge might be addressed either at the client, the server, a component that acts as an intermediary, or a combination of these. With the first, the client might be forced to re-discover the object and download its new proxy. In a specific implementation, a client might contact the UDDI registry [6] to re-discover a new copy of the WS (or a semantically equivalent WS). However, this approach has a limitation inherited from UDDI specifications. UDDI registries are replicated in different locations (forming a "UDDI cloud") and updates might take hours to propagate to all registries [17]. An extension to UDDI, Active UDDI [17], deploys itself as an additional WS that contains the UDDI WS and processes unavailability messages issued by different clients. It uses this information to conclude the availability state of a WS. However, this information is maintained per single registry basis and might require a longer time to propagate to other registries. Also, this infrastructure requires a distributed consensus to prevent faulty or malicious clients from affecting the state of a WS. Achieving such consensus requires centralized control which may introduce a single point of failure.

With a server-side solution, when a WS migrates from one node (say N1) to another (say N2), N1 maintains the forwarding address of this WS. When a client proxy references this WS on N1, N1 redirects this remote proxy to N2. This process might repeat recursively if the object migrates several times. This becomes cumbersome when the WS migrates from N2 back to N1. Somehow, the information must be propagated in a manner that prevents an indefinite referral of a client in search of a migrating WS.

Alternatively, the WS might publish its new location at a well-known address that enables its proxy to quickly locate its new location. This third alternative requires an external entity, a DeW registry, to resolve the location of a migrating WS. We describe the DeW registry as a general purpose infrastructure component that offers its functionality to a variety of services, e.g., a flight information management system, a document management system, etc.

DeW treats object migrations as a category of anticipated Internet changes, identified as exceptions

[2] We assume a query is a simple lookup. The centralized server continues to process complex queries, e.g., joins, aggregates, etc.

[14, 12, 19, 11]. We use the following definition of an exception in this study: it is a union of "exceptional case" or "unusual event" [19, 11]. An exception handler (or handler) is the code for one or several exceptions. The primary role of a DeW registry is to serve as a well known contact address for (a) WSs to publish exceptions and their handlers, and (b) proxy objects to download these handlers upon encountering an exception. In our example, the migrating WS publishes the handler for a predefined exception named "WebServiceNotFound" and registers it with the DeW infrastructure. When a proxy object attempts to invoke a method on a remote method of this WS, it contacts N1 and encounters the "WebServiceNotFound" exception, contacts the DeW infrastructure to download the newly registered handler, instantiates and executes this handler to obtain N2 as the contact server and recovers from this anticipated Internet change.

DeW employs the concept of exceptions for several reasons. First, it is available in a number of popular programming languages. Second, from a historical perspective, the use of exceptions has been multifaceted, evolving towards failure handling which is the focus of this study. In particular, several pioneering studies define exceptions as either an error or implementation insufficiencies that become part of normal exit or return [14, 12]. Exceptions have been embedded into both programming languages, e.g., Ada [24], C++ [33], Java [15], C#[1], etc., and operating systems, e.g., Medusa [25], Chimera [32], Windows NT [22], Mach [3], etc. Exceptions have also been applied to real time systems for detecting and handling timing errors [31]. The DeW framework applies exceptions in support of distributed, scalable, data intensive WSs middlewares. While it builds upon exceptions in programming languages (and appears similar to exception embedding in an operating systems), it is novel because it (a) enables exception handlers to evolve and change over time, (b) is application neutral, i.e., assumes a distributed environment where the infrastructure to store and retrieve handlers might be independent of the publisher of the handler and the proxies that download and execute the handler, and (c) provides physical location independence.

DeW is different from the Active UDDI framework [17] because it makes updating the location information the responsibility of the WSs. Active UDDI requires the client to contact the "active service" as soon as it encounters WS unavailability. The "active service" is an additional WS that contains UDDI web service. In addition to regular UDDI query/publish message, "active service" processes WS unavailability messages to update the status of WS contained in the UDDI registry.

The rest of this paper is organized as follows. Section 2 provides an overview of the DeW framework and details the DeW infrastructure including the design and implementation of proxy objects. Section 4 offers brief conclusions and future research directions.

2. DeW Framework

Our environment consists of (a) many service providers, and (b) one DeW infrastructure. The service providers maintain many WSs that are scattered across many nodes. To simplify discussion, and without loss of generality, assume the DeW infrastructure is administered by an entity separate from the service providers[3]. A service provider publishes its WSs and their DeWSDL files. Similar to WSDL, a DeWSDL is used to generate the proxy for a WS. It is an extended WSDL and is detailed in Section 2.1. There are no design restrictions on how a service provider authors their WSs. We do advocate a design template for a proxy object, described in Section 2.1, to solidify the paradigm. However, a service provider is free to design their proxy object using other alternatives.

When migrating a WS from one node to another, either the repository's middle-ware or the WS itself registers a new handler with the DeW registry to provide client proxies with the new location of the migrating WS. The proxy object for this WS is authored to raise a "WebServiceNotFound" exception if its referenced WS is no longer available (Section 2.1 explains how a proxy discovers WS unavailability). Once this exception is raised, it contacts DeW with the encountered exception and its referenced WS id. The DeW registry responds with the handler for this exception. The proxy object instantiates and executes this handler which enables it to recover from this anticipated Internet change to communicate with the WS.

In our example, the handler is simple because it might update certain variables in the client proxy to resolve the new address for this WS. Generally speaking, the handler might be sophisticated enough to replace the entire proxy object. This enables a service provider to deploy a new version of a WS (along with a new proxy object) with minimal disruption of service. Of course, in order to minimize service disruptions, the newly deployed proxy and service must be backward compatible with the prior versions.

[3] At the other end of the spectrum, each service provider deploys their own DeW infrastructure. This is beyond the focus of this study.

Both a WS and its proxy object must be able to contact the DeW storage subsystem, resulting in the following two requirements. First, the DeW infrastructure must be both scalable and high performance. Second, the environment must be operational at all times. If all DeW registries fail then: (a) the proxy objects will not be able to recover from a DeW exception, and (b) the WS (or WS administrators) will not be able to register their updated handlers. There are several alternatives implementations in support of this vision, ranging from a decentralized parallel database management system to a system based on IPv6's Anycast protocol. In [2], we describe two different implementations using 1) a structured peer-to-peer (P2P) system, and 2) one based on the Anycast protocol.

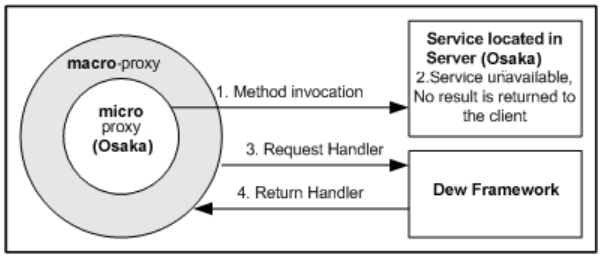
The current design of the DeW framework requires it to be passive in the following sense: when there are no faults, DeW is not involved in the communication between the client and its referenced WSs. This eliminates the overhead of heart-beat [29] messages. DeW is based on a software paradigm where desired functionality or behavior is achieved by raising an "exception", and then "handling" the raised exception.

With DeW, the handler stored for each exception might be code, data or both. For example, in the flight service application, once a failure for one specific service occurs, the proxy raises a "WebServiceNotFound" exception and downloads its handler from DeW. The handler may contain a list of locations a replica of the referenced service is running, and the applicable code that chooses a location using a pre-specified criteria. The result of handling the exception is choosing another location that hosts the same service and re-invoking the method automatically. DeW enables a service provider to update the handler for an exception with no service disruption.

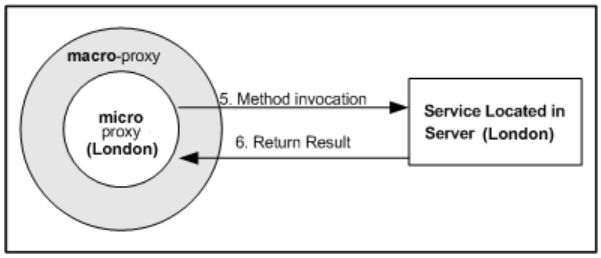
In the following subsections we detail DeW's components using an example of remote services for airline flight reservation systems, see Section 1.

2.1. Dew Proxy and DeWSDL

The purpose of the proxy is to hide the remote invocation details. In other words, with a proxy, the user is not aware of remote invocation. WS proxy is an important component of DeW. Typically, the proxy's functionality is restricted to initiate a remote invocation in either a synchronous or an asynchronous manner and to provide results back to the client. If a failure is encountered, the proxy fails, forcing its client to re-issue the request manually. With DeW, we employ a two-layer proxy, termed macro and micro proxies. The micro proxy is application specific and implements the necessary functionality of a remote service. The macro proxy intercepts remote invocations of the micro proxy and implements timeouts in order to prevent an indefinite wait-time for the remote server. The macro proxy is also responsible for raising DeW exceptions in the presence of time-outs, downloading the handler from DeW storage subsystem, and its execution to recover from the failure. The exception handler for a macro proxy might even replace the micro-proxy with one that references another remote service.



1.a Macro proxy catches an exception caused by an unsuccessful invocation



1.b After processing the exception handler, connection is re-established

Figure1. Macro and Micro proxies and their relationship

To explain the functionality of DewProxy, imagine the situation where a traveling agency worker is using a system that utilizes the airline reservation system of Section 1. Consider a remote service for the flight from New York to Los Angeles that is replicated across three different nodes that are geographically located at: 1) London, UK, 2) Los Angeles, USA, and 3) Osaka, Japan. Let's assume that the airline application downloads the proxy for the service in Osaka (the details of how an application discovers or downloads a proxy for a service is not the focus here, see [29, 6] for details), and the service either fails or becomes network unreachable. Without DeW, if the travel agent tries to inquire about the available seats, s/he will get a message indicating the failure. With DeW, the macro proxy will start a timer prior to invoking the remote method on the Osaka server. A successful method invocation will return before the timer expires. However, if the timer expires before obtaining the results, the method invocation is considered to be

unsuccessful[4]. If this happens the macro proxy raises an exception and downloads the handler from DeW storage subsystem as in Figure 1.a. It executes the handler within the macro proxy and replaces the Osaka "micro" proxy to point to another replica of the same service, say the one in London, and automatically re-invoke the method as in Figure 1.b.

Due to the popularity of WSs, many tools are available to generate the proxy of a WS from its WSDL file. WSDL is an XML specification that defines, among other things, methods exposed by a WS, their inputs, outputs and exceptions. When communicating with a WS, the proxy might encounter an exception in two possible scenarios: 1) when obtaining unexpected results, 2) when no result is obtained from the service. The first is already defined and supported using WSDL specifications in the <fault> element under the <operation> element. To support the second scenario, we use an extended WSDL, termed DeWSDL.

The extensions we provide only impacts the <operation> element of WSDL which defines the methods exposed by the WS. The DTD of the extended WSDL is as follows:

<!ELEMENT exceptionlist (dewexception*)>
<!ELEMENT dewexception(ID, timeout)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT timeout (#PCDATA)>

To generate DeW-compatible proxies, the <operation> element in the DeWSDL file must include an <exceptionList> element which in turn lists a set of <DewExceptions> elements. Using this syntax, the

WS programmer specifies the kind of exceptions s/he wants the DeW framework to handle.

Existing tools that consume WSDL to produce the proxy software (using Java, C#, etc.,) are not flexible enough to support the concept of micro and macro proxy. To maintain the conveniences of automatic generation of proxies, we have developed a tool that consumes the DeWSDL of the WS and produces a DeW-compatible proxy, i.e. containing both the macro and micro proxy.

```
<operation name="doGetCachedPage">
      <input message="typens:doGetCachedPage"/>
      <output message="typens:doGetCachedPageResponse"/>
</operation>
```

2.a excerpt of Google search WSDL

```
<operation name="doGetCachedPage">
      <input message="typens:doGetCachedPage"/>
      <output message="typens:doGetCachedPageResponse"/>
      <exceptionlist>
            <dewexception>
                  <ID> ServiceNotReachable </ID>
                  <timeout> 10 </timeout>
            </dewexception>
      </exceptionlist>
</operation>
```

2.b excerpt of Google search DeWSDL

Figure 2. WSDL and DeWSDL

Note that extensions introduced in DeWSDL are backward compatible, meaning that DeW-incompatible tools need not understand them. Of course, if one employs today's commercial tool to process DeWSDL, the resulting proxy will not be able to contact DeW. Figure 2.a shows a regular WSDL excerpt and 2.b shows its DeWSDL equivalent.

2.2 DeW Storage Subsystem

One may implement DeW as a centralized storage subsystem to manage all the exception names and their handlers. Another alternative is to employ a P2P Overlay network that provides fault-tolerance and a well-balanced environment by utilizing Distributed Hash Tables (DHTs). Example systems include CAN [27], Pastry [28], OceanStore [9], etc. In DeW, we use CAN to store each pair <<service-id, exception-name>,handler>. P2P Overlay networks store <key,value> pairs where a key is a composite consisting of <serviceid, exception-name> and its value is a handler. The hash function maps a key to a peer in the network where the value for that key is stored. Once a WS proxy encounters an exception, it contacts DeW's storage subsystem with the <service-id,exception-name>, which we use as a key, and the exception handler is located and downloaded to the client side for processing.

The handler might be code, data or both. For example, in the flight reservation application, a handler stores a list of locations (XML formatted data) where a service is replicated and the code that chooses one location based on a certain criteria. The result of downloading and executing this handler is to choose an alternative location that hosts the same service and re-invoke the method seamlessly. DeW does not have any restrictions on the format used to store the binary data. However, we suggest encapsulating the data and code in XML format. This is preferred for two reasons. First, it utilizes the hierarchal structure of XML. Second, it simplifies both querying and manipulation of data.

With anycast, all the DeW registries listen on a well known anycast address. This address is used by the proxy objects and the WS. Anycast simplifies the task of introducing an additional node into the DeW registry

[4] Timeout occurs when SOAP messages are exchanged using TCP. However, if the WS is hosted using a web server (i.e. SOAP message is transmitted using HTTP), the web server will return a status code, similar to 404, indicating that the requested WS is not found. In this case the method invocation will not result in a timeout, forcing the proxy to parse the status code to detect the unsuccessful method invocation.

cloud; load this node with the necessary DeW software and deploy it to listen on the anycast address. Anycast enables a source node (a service provider or a proxy object holder) to transmit datagrams to a single DeW node, out of a group of DeW nodes. Upon receiving a DeW node reply, the sender discovers the unicast address of the DeW node to establish a stateful connection, e.g., TCP, with that node[5]. While routers that support anycast might not be widely deployed at the time of this writing, we envision it to become common place in the near future because it is part of the IPv6 specification suite [16]. This specification is currently supported by tens of router manufacturers, e.g., 3Com, Cisco, Hitachi, Juniper, etc. As these routers are deployed, the DeW framework using anycast becomes more viable.

2.3 Security

Faulty or malicious WSs impact only their own clients and cannot impact the operation of the DeW framework. However, malicious servers joining DeW's storage subsystem may adversely impact its operation by either (1) modifying handlers stored in them (compromising data integrity) or (2) removing themselves from the P2P network to render the handlers stored in them unavailable. DeW addresses both issues. Consider each in turn. First, the DeW storage subsystem is deployed with multiple realities (see design improvements in [27]) that replicate certain <key, value> pair by storing them across multiple nodes, increasing the availability of handlers in the presence of storage subsystem node failures.

Second, we require the use of message digest and digital signing algorithms to preserve the integrity of handlers. Before storing handlers, a WS computes a message digest for the handler using one of message digest algorithms, e.g., MD5. This message digest is a one-way hash. Next, the WS signs (encrypts) it with its private key which is typically the same as the private key of the publisher of the WS. DeW stores the handler data and its signed message digest.

When a proxy downloads a handler, it (1) computes the digest for the handler, and (2) decrypts the signed message digest using the WS owner's public key. If these two obtained values match then the handler is valid. Otherwise, the proxy flags this handler as erroneous and discards it. Next, it contacts the DeW storage subsystem with a different "entry" node, re-downloads the handler, and repeats the process. WS Owner's public key is obtained from the owner's digital certificate. A Public Key Infrastructure (PKI) facilitates this key exchange.

When a WS migrates, it must modify the handler. This requires the WS to re-compute the message digest of the modified handler and encrypt it. Next, the WS transmits the modified handler and its encrypted digest to DeW for storage. Most programming languages include sufficient capabilities to integrate cipher text manipulation in different components of the WS. Examples include Sun's Java Cryptography Extension (JCE), Microsoft's System.Security.Cryptography namespace in .NET framework [18], etc.

2.4 Multiple Concurrent Updates

An environment may construct multiple copies of a WS to balance system load. It is possible for these WSs to migrate at the same time and attempt to publish new handlers simultaneously. We describe two alternative approaches to serialize these concurrent updates. The first requires DeW to employ locking while the second utilizes incremental updates. We describe each in turn.

The lock-based approach might be implemented in a variety of ways. In the following, we describe one version that employs eXclusive (X) and Shared (S) locks. Next, we describe the advantages and disadvantages of this approach and its variations. With the lock-based approach, DeW (a) distinguishes between read and update requests, (b) supports X-locks to serialize updates, and (c) uses Shared locks to prevent a proxy to read a handler that is in the process of being updated. Read requests are issued by proxy objects while updates are performed by migrating WSs. A WS must lock a handler in X mode prior to updating it and unlock the handler once its update completes. When two or more WSs compete for an X-lock on the same handler, one is granted the lock while the others block. Shared locks are compatible with one another allowing multiple read requests to proceed as long as there is no X lock, i.e., DeW is not writing the referenced handler.

When DeW grants an X-lock (S-lock) to a WS (proxy), the node containing that WS (proxy) might fail before the WS (proxy) releases its lock. DeW might employ different techniques to detect and resolve such scenarios. Here, we describe two alternative possibilities. With the first, DeW times out if a WS does (proxy) not release its lock after a pre-specified threshold. The WS (proxy) may specify this threshold when issuing a X-lock (S-lock). Another alternative is for the WS to use heart-beat messages [29] to detect if the node containing the WS (proxy) has failed.

[5] See IETF specification [26] for how to convert an anycast exchange into a TCP connection.

The overhead of setting S-locks might be too high and one may configure the proxy objects to not request such locks when downloading a handler. This may cause a proxy to read either an obsolete handler or a handler that is partially over-written. An obvious draw-back is that the proxy may encounter another failure and be forced to contact DeW again. This process might repeat several times until a new handler is finally published. A deployment of DeW would be useful to decide which approach is more suitable. We believe a choice is dependent on the needs of a target application and its frequency of updates to handlers.

With DeW supporting data as a possible handler, an alternative to locking is for a WS to propose changes to this data. For example, the handler for an exception might be a list of possible servers. The following example shows a handler that identifies three servers containing a handler:

<Servers>
<Server id="1">125.23.22.14</Server>
<Server id="2”>155.32.42.94</Server>
<Server id="3">105.81.66.90</Server>
</Severs>

If the replica on IP address 125.23.22.14 migrates to 125.23.22.27, then it issues (a) the removal of 125.23.22.14, and (b) insertion of 125.23.22.27. Each of these is termed a delta [13]. These deltas might be encrypted and signed, see Section 2.3. Without the private key of a service provider, DeW cannot apply these changes to a handler. When a proxy requests the handler, it is provided with both the handler and its deltas. The proxy decrypts both, and applies deltas to the handler to obtain its most up-to-date version. With multiple replicas of a WS migrating simultaneously, each proposes a delta and DeW maintains these deltas. Over time, a handler may consist of a chain of deltas with C elements: $\delta_1, \delta_2, ..., \delta_C$. When a WS contacts DeW to register a new delta, DeW may ask the WS to merge the handler with the first k deltas, $k<C$. This enables DeW to replace the old handler and its k deltas with the newly computed handler.

This incremental change driven paradigm has the following constraints. This means, a protocol must exist that enables DeW to request a WS to merge a handler with its k deltas. This means the programmer is responsible for specifying the necessary logic that enables: a) a WS to propose deltas, b) a proxy to merge a handler with a sequence of deltas, and c) a WS to apply k deltas to a handler, sign the new handler and send it back to DeW. Second, the design of handler and its deltas are application specific. Thus, there might exist scenarios where deltas are not associative. In such scenarios, DeW must defer to a lock based approach. In general, DeW may support both paradigms to enable the service provider choose the appropriate paradigm.

3. DeW's and Software engineering

An application's life cycle might consist of several software development phases. The exact nature of each phase and its relationship to other phases depends on the assumed software engineering paradigm. The software engineering literature includes several paradigms such as: waterfall model [4], spiral mode [5], etc. Most, if not all, include analysis, design, Implementation, and maintenance phases. Figure 3, for example, shows the phases of waterfall model. Developing a DeW-enabled application impacts several of these phases. In the Design/Analysis phase, the WS developers must consider the impact of DeW's framework. They must specify the alternative Internet exceptions along with their handlers. During the Coding/Implementation phases, DeW requires slight modification to the way a service proxy is implemented. This modification minimizes the burden on the remote service programmer by requiring s/he to include a software library, a shared DLL, that defines the Internet exceptions of an authored proxy to be of type "DeWException". It is the programmer's responsibility to anticipate the possible Internet exceptions, identify them as such, and author handlers that enable the proxy to recover from such failures. Moreover, the programmer must consider whether the WS or its target environment registers new handlers in the presence of location updates. Thus, software correctness is the programmer's responsibility.

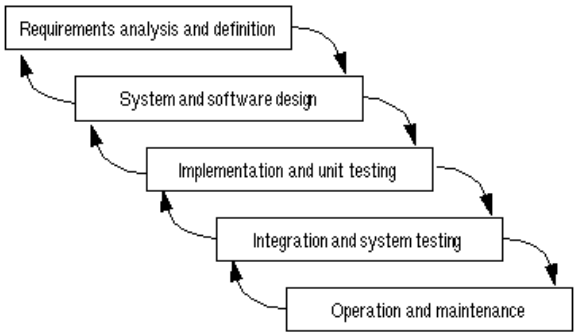


Figure 3. Phases of waterfall model

4. Conclusion and Future Research

This paper motivates the DeW framework in support of physical location independence. We have implemented a prototype of DeW using the C# programming language [20]. Our target application is a simple service that represents a scientific document as

a WS. Each WS represents a document, exposing the following methods: getTitle, getAuthor, getDate, getAbstract, getText and getImages. We deployed our implementation in an intranet setting. DeW's CAN network may consist of an arbitrary number of nodes and we have experimented with as many as six nodes. We deployed several services in our private network, and tested different failure scenarios including service failure, migration and unavailability. Using DeW, as long as one operational service replica exists, the DeW framework is able to seamlessly recover from the failure and direct the client proxy request to the properly functioning service. In the immediate future, we are exploring the integration of DeW with WS frameworks such as GXA [21], BPEL4WS [10], and XL [8, 7].

5. References

[1] B. Albahari, P. Drayton, and B. Merrill. C# Essentials. O'Reilly, 2001.

[2] E. Alwagait and S. Ghandeharizadeh. DeW: A Dependable Web Services Framework. Technical Report dblab-2003-2, USC, March 2003.

[3] D. Black, D. Golub, K. Hauth, A. Tevanian, and R. Sanzi. The Mach Exception Handling Facility. Technical report, School of Comp Sci, Carnegie Mellon University, 1988.

[4] B. Boehm. Software engineering. In IEEE Trans. on Comp., vol. C-5, no. 12, pp. 1226-1241, Dec. 1976.

[5] B. Boehm. A spiral model of software development and enhancement. Computer, 21(5):61–72, 1988.

[6] UDDI Community. UDDI Version 2.0 Data Structure Specification. UDDI Organization, June 2001.

[7] D. Kossmann D. Florescu, A. Grnhagen. XL: An XML Programming Language for Web Service Specification and Composition. In WWW2002, International World Wide Web Conference, May 2002.

[8] D. Kossmann D. Florescu, A. Grnhagen. XL: A Platform for Web Services . In Conference on Innovative Data Systems Research (CIDR), January 2003.

[9] J. Kubiatowicz et al. OceanStore: An Architecture for Global-scale Persistent Storage. In Proceedings of ACM ASPLOS. ACM, November 2000.

[10] J. Klein F. Leymann S. Thatte F. Curbera, Y. Goland and S.Weerawarana. Business Process Execution Language for Web Services, Version 1.0. July 2002.

[11] A. Garcia, C. Rubira, A. Romanovsky, and J. Xu. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Elsevier Journal of Systems and Software, 59(2):197–222, November 2001.

[12] N. Gehani. Exceptional C or C with Exceptions. Softw. Pract. Exper., 22(10):827–848, 1992.

[13] S. Ghandeharizadeh, R. Hall, and D. Jacobs. Design, Implementation, and Application of Heraclitus[Alg,C]. ACM-Trans on Database Systems, 21(3):370–426, 1996.

[14] J. Goodenough. Exceptional Handling: Issues and a Proposed Notation. Communications of the ACM, 18(12):683–696, 1975.

[15] J. Gosling. Java: an Overview. Sun Microsystems, February 1995.

[16] R. Hinden and S. Deering. IP Version 6 Addressing Architecture. Internet Engineering Task Force (IETF), RFC2373, July 1998.

[17] M. Jeckle and B. Zengler. Active UDDI - An Extension to UDDI for Dynamic and Fault-Tolerant Service Invocation. In 2nd Annual International Workshop of the Working Group on Web and Databases of the German Informatics Society (GI),Thuringia, Germany, October 2002.

[18] B. LaMacchia, S. Lange, M. Lyons, R. Martin, and K. Price. .NET Framework Security. Addison Wesley Professional, April 2002.

[19] J. Lang and D. Stewart. A Study of the Applicability of Existing Exception-Hnadling Techniques to Component-Based Real-Time Software Technology. ACM Transactions on Programming Languages and Systems, pages 274–301, March 1998.

[20] Microsoft Corporation. Microsoft .NET Framework.

[21] Microsoft Corporation. Global XML Web Services Architecture (GXA), 2002.

[22] S. Niezgoda, S. Holt, and D.Wojciech. Some Assemply Required: NT's Structured Exception Handling. BYTE, 18(12):317–322, 1993.

[23] NIST. FIPS 180-1, Secure Hash Standard, April 1995.

[24] Department of Defense. Reference Manual for the Ada Programming Language. US Department of Defense, 1983.

[25] J. Ousterhout. Partioning and Cooperation in a Distributed Multiprocessor Operating System: Medusa. Carnegie-Mellon University, 1980. Ph.D. Thesis.

[26] C. Partridge, T. Mendez, and W. Milliken. Host Anycasting Service. Internet Engineering Task Force (IETF), RFC1546, November 1993.

[27] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content Addressable Network. In ACM SIGCOMM, 2001.

[28] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In International Conference on Distributed Systems Platforms (Middleware), November 2001.

[29] T. Schroeder, S. Goddard, and B. Ramamurthy. Scalable Web Sever Clustering Technologies. IEEE Network, pages 38–45, May 2000.

[30] F. Sommers, S. Ghandeharizadeh, and S. Gao. Cluster-Based Computing with Active, Persistent Objects on the Web. In IEEE 3rd International Conference on Cluster Computing, October 2001.

[31] D. Stewart and P. Khosla. Mechanisms for Detecting and Handling Timing Errors. Communications of the ACM, 40(1), 1997.

[32] D. Stewart, D. Schmitz, and P. Khosla. The CHIMERA II Real-Time Operating System for Advanced Sensor-Based Control Applications. IEEE Transactions on Systems Man and Cybernetics, 22(6), 1997.

[33] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1997.