Caching of XML Web Services for Disconnected Operation

Venugopalan Ramasubramanian and Douglas B. Terry

Microsoft Research
1065 La Avenida
Mountain View, CA 94043

Abstract

Caching can permit mobile applications to utilize XML Web services in the face of intermittent connectivity. However, Web services present a number of challenges to cache managers since they have generally been designed without regard to caching and hence provide little support. The WSDL description of a Web service, for instance, specifies the SOAP message formats needed to invoke service operations but does not indicate which operations modify the service's persistent state or will likely produce different results on different invocations. We developed tools for generating custom cache managers that tailor their behavior based on annotations added to a Web service's WSDL specification. An experiment to transparently interject a cache between existing XML Web services and their clients running on mobile devices demonstrated that disconnected operation could be achieved without modifying services or their applications.

1 Introduction

Web services are emerging as the dominant application on the Internet. The Web is no longer just a repository of information but has evolved into an active medium for the providers and consumers of services. Services on the Web are provided and consumed by a wide variety of entities: individuals provide peer-peer services to access personal contact information or photo albums for other individuals; individuals provide services to businesses for accessing personal preferences or tax information; a plethora of Web based businesses provide consumer services such as travel arrangement (Orbitz), shopping (eBay), and electronic mail (Hotmail) and, in addition, several business-to-business (B2B) services such as supply chain management form important applications of the Internet. While these services are currently being provided through static or active Web pages, they are evolving into XML Web services designed for programmatic rather than human access. For example, MapPoint.Net, provides maps and location services for incorporation into other Web sites and applications. Thus, XML Web services are the building blocks for constructing a new generation of Web applications that leverage existing investments in Web technology.

One of the key requirements for the success of Web services is universal availability. Web services tend to be accessed at all times and at all places. Clients employ a wide range of devices including desktops, laptops, palm or hand held devices, and smart phones that are connected to the Internet using very different kinds of networks such as wireless LAN (802.11b), cell phone network (WAP), broadband network (cable modem), telephone network (28.8 kbps modem), or local area network (Ethernet). Frequent disconnections and unreliable bandwidth characterize some of these networks. The availability of Web services is thus a significant concern to consumers using mobile devices and working in different kinds of mobile networks.

A good solution to improve availability of Web services should be transparently deployable and generally applicable. Transparent deployment means that the solution must not require changes to the implementation of the Web services, either to the server and client side modules or to the communication protocol between them. The growth in the number of Web services has been phenomenal and hence applying changes to existing Web services is an improbable proposition. For the same reason, the solution should also be scalable and general enough to apply to all the Web services. Thus building specialized components to handle disconnections for each Web service would be extravagant. A good solution would be applicable to all Web services and would involve interposing storage and computation transparently in the communication path of the client and the server without modifications to Web service implementations on the client or the server.

This paper presents an architecture that supports continued access to Web services from mobile devices during disconnections. This architecture provides a client side cache that mimics the behavior of Web services to a limited extent. Caching satisfies both the required characteristics of transparent deployment and general applicability. This cache architecture strictly conforms to the XML based standards for Web services developed by the W3C (World Wide Web Consortium). The cache is transparent to both the client and server components of the Web services. Hence no changes are required to the implementation and the communication protocol of the Web service, and the architecture can be applied to already existing Web services that conform to W3C standards. We have built a prototype of this architecture and applied it to improve the availability of Microsoft's .NET My Services set of Web services. The implementation of this prototype and a case study application are detailed in this paper.

This paper is organized into the following components. Previous work relevant to this problem is discussed in Section 2. Section 3 presents a brief description of the XML based standards for Web services. Section 4 details the issues associated with building a cache for Web services. Sections 5 and 6 contain a detailed description of the Web services cache architecture that we developed. In Section 7, we discuss an application of our architecture to improve the availability of the *MyContacts* Web service that forms a part of the .NET My Services. We conclude in Section 8.

2 Related Work

A variety of systems have employed caching on mobile devices in support of disconnected access to files [16][23], databases [1][5][22], objects [19][26], and Web pages [6][7][8][13]. This paper presents the first significant exploration of caching of XML Web services. The basic architecture of all of these systems, including ours, is fairly similar, consisting of a cache to store read/query results and a write-back queue for updates [18]; caching web clients invariably use a proxy-based architecture [20]. The differences lie in what is being cached, how the cache is managed, and the degree of cooperation between clients and servers. XML Web services present new challenges due to the diverse set of operations exported by such services as well as their lack of involvement in the caching process.

Most modern commercial Web browsers allow users to access cached pages while offline but pay little attention to cache consistency. More advanced techniques for caching Web pages, on both fixed and mobile devices, have focused primarily on maximizing cache hit rates while preserving some degree of cache consistency. Web caches map URLs to HTML pages and need worry about only one operation, namely the HTTP GET operation. Cache managers rely on directives provided by Web servers that indicates whether a page is cacheable and for how long. XML Web services are active entities and hence, passive caching as defined by HTTP [12] is unsuitable. The default HTTP cache directive for Web service responses is *no-cache*.

Caching to improve availability of file systems and database systems is a well-explored and widely used technique. However, Web services differ considerably from these traditional distributed systems. Both file systems and database systems have well-defined client interfaces. For example, a file server exports standard operations like *read*, *write*, *open* and *close* to clients. A cache manager need only implement this interface whose semantics are well understood. In contrast, each Web service exposes its own distinct interface to clients.

Replication of data onto mobile devices, like caching, can be done to provide high availability in the face of intermittent connectivity [16][17][23][32]. While XML Web services almost always manage persistent databases internally, simply replicating that data is not useful. Web services encapsulate their data in critical business logic. The attraction of Web services is in allowing client applications to leverage this high-level logic. Thus, the large body of previous work on replicated file systems and databases is not directly applicable to Web services.

Distributed object systems have also supported caching or replication to permit disconnected operation and improve performance [10][19][24][26][28]. Web services can be viewed as objects in that they both export a variety of operations through published interfaces. Replicated object systems, however, have assumed that objects, including code and data, can be cached in their entirety. Such an assumption may not apply to mobile devices with limited resources. More importantly, a Web service's code, i.e. business logic, is often treated as proprietary and is generally owned by different parties than those accessing the services. Replicating such code onto mobile clients is therefore infeasible.

This is what caused us to pursue a different path for accessing Web services from intermittently connected devices, namely request/response caching. Others have suggested this same approach and identified some of the issues [28][30][31], but we know of no other attempts to attack the challenging practical problems in caching Web services for disconnected operation and to produce a working implementation.

```
<?xml version="1.0" encoding="utf-8"?>
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/" xmlns:m=http://schemas.microsoft.com/hs/2001/10/myContacts
xmlns:c=http://schemas.microsoft.com/hs/2001/10/core xmlns:mp="http://schemas.microsoft.com/hs/2001/10/myProfile" >
  <s:Header>
    <licenses xmlns="http://schemas.xmlsoap.org/soap/security/2000-12">
      <c:identity> <c:kerberos>3240</c:kerberos> </c:identity>
    </licenses>
    <path xmlns="http://schemas.xmlsoap.org/rp/">
      <action>http://schemas.microsoft.com/hs/2001/10/core#request</action>
      <to>http://microsoft-m3we4f.microsoft.com</to>
      <fwd><via /></fwd><rev><via /></rev>
      <id>b55528a4-5d63-49f1-87a2-5fab8d76f658</id>
    </path>
    <c:request service="myContacts" document="content" method="insert" genResponse="always" >
      <key puid="3240" instance="1" cluster="1" />
    </c:request>
  </s:Header>
  <s:Body>
    <c:insertRequest select="/m:myContacts/m:contact[mp:name/mp:givenName = 'Joe']/mp:emailAddress" >
      <mp:email>joe@idontknow.com</mp:email>
    </c:insertRequest>
  </s:Body>
</s:Envelope>
```

Figure 1: Example SOAP message carrying insert request for *MyContacts* Web service.

3 XML Web Services

Web services consist of a service provider and multiple consumers based on the client-server architecture. Each Web service uses a custom communication protocol for the clients to access the servers. The most common access pattern for a Web service consists of requests and responses. The client sends a request message that specifies the operation to be performed and all relevant information to perform the operation, to the server. The server performs the specified operation and replies with a response message. The actions carried out by the server might result in permanent changes to the sate of the server.

Essentially, Web services provide RPC like interfaces to the client. For example, *MyContacts*, one of the .NET My Services [27], is a Web service that allows users to maintain access information such as names, addresses and phone numbers of their contacts. The *MyContacts* Web service exports operations to insert, delete, replace and query portions of this contact information. Each of these operations takes input parameters (the query string) and produce output (query response or success status) while making permanent state changes at the server. Each Web service provides its own custom interface that could be vastly different than those provided by other Web services. For example, a travel Web service would provide operations to search for airfares, reserve and buy tickets and look-up itinerary.

The World Wide Web Consortium (W3C) has recommended a set of standards for Web services based on the Extensible Markup Language (XML) [3] with the support of several leading corporations including IBM and Microsoft. These XML based standards provide globally recognizable protocols for discovering, describing and accessing the custom interfaces of Web services [4][10]. This standard consists of two important components: SOAP (Simple Object Access Protocol) and WSDL (Web Services Description Language).

The Simple Object Access Protocol (SOAP) [14][15] specifies a standard for sending messages between different entities of a Web service. SOAP messages are XML documents that are transported from one *SOAP-node* to another. For Web services, the SOAP-nodes could be either the client or the server. Each SOAP message consists of an outer-most element called the *envelope*. The envelope consists of two elements: a mandatory *body* element, and an optional *header* element. The body element carries the main content of the message. For the case of a request message, it would carry the name and parameters of the operation to be performed. The header element consists of multiple header blocks, each containing meta-information for the receiver or intermediary nodes. The header blocks are used to specify additional useful information such as password for authentication. The SOAP message in Figure 1 shows an example request message from the client to the server for the *MyContacts* Web service.

The Web Services Description Language (WSDL) [9] is a standard used to provide descriptions of Web services. The WSDL document for each Web service completely describes the custom interfaces provided by that Web service to clients. This document can be used by program development tools, such as Microsoft's Visual Studio .NET, to automatically generate proxy stubs that encapsulate the remote Web service as a local object on the client. The WSDL document lists the names of the operations provided by the Web service as well as the format of SOAP messages used to communicate between the client and the server. A complete description of the data types of the parameters to be passed to each operation or received as responses is also provided in the WSDL document of a Web service. Thus WSDL is used to provide adequate description of the varied interfaces provided by the Web service.

The caching architecture presented in this paper supports all Web services that conform to the W3C standards, provide WSDL descriptions of their interfaces and use SOAP messages for communication with clients. The implementation of this architecture assumes that the SOAP messages are transported using HTTP (Hyper Text Transfer Protocol) as the application layer protocol since the SOAP-RPC recommendations are currently complete only for HTTP. Accordingly, the request for an operation from a client is carried by a HTTP request message and the response from the server is carried by the corresponding HTTP response message.

4 Issues in Caching Web Services

To study the suitability of caching to support disconnected operation on Web services, we conducted an experiment in which a caching proxy was placed between Microsoft's .NET My Services and the sample clients that ship with these services. The .NET My Services were chosen for this experiment because they are the only example we could find of publicly available, non-trivial XML Web services that support both query and update operations.

We built a HTTP proxy server as defined in the HTTP protocol standard [12] and deployed it on the client device. In this case, the client device was a laptop running Windows XP. All HTTP messages originating at the client, including those generated by Web clients and Internet browsers were made to pass through the proxy server. The proxy server acts as a simple tunnel for all HTTP packets that are not SOAP messages.

A cache for storing SOAP messages was added to the proxy server. This cache stores SOAP messages received in response to SOAP requests. All cache policies for expiration and replacement were implemented as recommended in the HTTP standard [12]. During the experiments, this cache was used only to store HTTP packets with SOAP messages as their entities. Whenever the network is connected, the SOAP request received would be sent to the server. The received SOAP response is stored in the cache associated with the request, replacing the old response if it had existed. In the case where the cache contained a previous response for the same request, the new SOAP response is compared to the previous response and the result of the comparison is recorded in a log file. These comparisons provided valuable insight into what kind of operations could be cached and what other operations affect the validity of the cached responses.

If the network is disconnected, the SOAP response stored in the cache (if present) is returned to the client and the SOAP request is stored in a write back queue. If a request has no cached response, the client times out waiting for the response, as in the normal case when a server is unreachable. All requests are stored in the write back queue for later replay since the cache manager cannot determine which requests modify the service state and which are simply queries. Whenever the connection to the Web service is restored, the queued up SOAP requests are played back to the server and the SOAP responses are stored in the cache.

We carried out this study using the .NET My Services published by Microsoft [27]. The .NET My Services Software Development Kit (SDK) contains a number of Web services such as *MyContacts* that allows users to store and retrieve address and phone information, *MyProfile* that allows users to store their personal information, and *MyFavoriteWebSites* that allows clients to manage favorite Web sites. The SDK also comes with several sample applications that call on these services. By running the sample applications, we performed various operations on these services having the network connected as well as deliberately disconnecting the network. Figure 2 illustrates the set-up of our experiments.

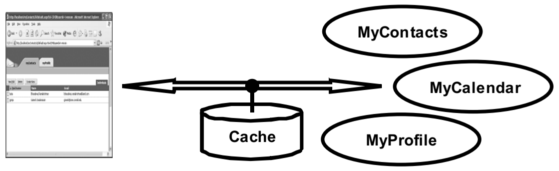

Figure 2: Experimental set-up to study the benefit of caching for Web services

These studies clearly highlighted the benefits of employing a Web service cache to support disconnected operation. In particular, the applications ran just fine while disconnected as long as the cache was preloaded. However, the experiment also exposed a number of issues that need to be handled in order achieve a significant improvement in the consistency and availability of offline access to Web services. The rest of this section elaborates on various problems associated with designing a client cache for Web services.

Playback and Cacheability

The diverse nature of Web services poses a major problem in identifying the semantics of the operations exposed by the Web service. In the case of file systems, the semantics of standard operations such as *read* and *write* are clearly understood. Results of the *read* operation can be stored in the cache while the *write* operations need to be played back to the server upon restoration of connectivity. On the other hand, Web services have very diverse interfaces that make it extremely difficult for the cache to understand whether a certain operation needs to be played back to the server and whether an old response from the cache is acceptable to the client.

At least two properties of an operation need to be recognized by the Web service cache in order to function effectively. An operation is said to be an *update* if execution of the operation makes permanent changes to the state of the server. An operation is said to be *cacheable* if subsequent execution of the operation with the same parameters produces the same response, provided that no update operation intervenes. Operations of a Web service could be both cacheable and updating while others could be neither or have one property without the other. For example, a request to query data is cacheable but generally is not an update. However, a query could also update the server state if the server needs to maintain a log of all requests. A request to get the current time is neither cacheable nor an update.

Consistency

A fundamental challenge faced by caching schemes in general and compounded by the diverse nature of Web services is providing basic consistency guarantees. When operating in disconnected mode, a cache manager cannot provide strong consistency guarantees since it does not have access to updates performed by other users. However, it can at least strive to provide cache results that are consistent with a user's own actions. In particular, an operation that is performed by the local user could change some of the results of the earlier requests that are stored in the cache. For example, in the *MyContacts* Web service, a request to change the telephone number of a friend would invalidate an earlier response querying the contact information of that friend. In order to maintain correctness, the earlier response in the cache would have to be either correctly modified or deleted from the cache. Otherwise, the cache may return an incorrect response if the query is again repeated during a network outage. For pre-existing Web services, understanding the correct consistency requirements, that is, the interdependencies between operations, is an extremely challenging issue.

User Experience

An important criterion for the assessment of a good technique to support disconnections is its effect on the user's experience. Ideally, a user should obtain the same experience when disconnected as when connected. However, achieving the ideal goal is not practically feasible, especially if the Web client is unaware of the existence of the cache.

There is a direct trade-off between the consistency guarantees assured by the Web service cache and the quality of the user experience during disconnections. By providing only weak consistency guarantees, the Web service cache can greatly improve the availability of Web services. For example, when the cache handles a request for an update operation, in addition to storing that request for future playback to the server, it could send a fake response to the Web client reporting the success of this operation. However, when the request is actually played back to the server upon reconnection, the server may decide to abort that operation for various reasons. On the other hand, guaranteeing strong consistency would not affect a user's experience but would prohibit the cache from employing certain techniques to enhance the service's availability.

Making the Web client application cache-aware can aid users during off-line access to Web services. In particular, the Web service client could appraise the user about disconnections and uncertainties in the execution of certain operations. However, modifying Web clients to add this reporting functionality may not be an easy task. Our study suggested the need to discover a standard mechanism for reporting important events to users that did not require substantial alteration of the Web client implementation.

Communication Protocol

Understanding the format of the messages exchanged between a Web service client and server can be another problem for a caching proxy. Despite using standard protocols, such as SOAP, Web services deviate considerably in the structure of their request and response messages. Even mechanisms for identifying the name of the operation being performed vary from service to service. For example, in Figure 1 showing an example SOAP request for the *MyContacts* Web service, the operation name, "insert", is one of the attribute fields of the request header. The operation name may have to be identified in a completely different way for a different Web service.

Correct comprehension of the message structure is also required for other fundamental reasons such as comparing requests and sending default responses. For example, each SOAP request message of the *MyContacts* Web service has a unique identifier in one of the header fields (see Figure 1). The cache manager must ignore this field during comparisons in order to correctly recognize similar requests. Otherwise, every request would be different and the cache would be rendered ineffective. When a cache receives a request for an update operation during disconnection, it is expected to return a meaningful response to the client in order to pretend that the service is available. If this operation is also cacheable, the cache can return a response stored earlier; if not, it needs to generate a response that conforms to the message format of that service.

Other Issues

The effectiveness of a cache depends on the similarity of future requests to the past requests. The cache can return stored responses only for those requests seen earlier by the cache. Hoarding techniques that preload the cache with responses of requests that are anticipated in the future can significantly improve availability [24]. However, selecting the right requests for hoarding requires the involvement of the user. Developing a standard mechanism for users to specify hoard requests that can be used by the Web service cache is an interesting challenge.

Security is another important issue that needs to be considered while building a cache for Web services. Web services often check the authentication of the messages and the authority of the users before performing operations. For example, the expiry of a Kerberos ticket might prohibit a user from accessing certain information. The cache manager may need to include mechanisms to perform these authorization checks before responding from the cache during disconnections. Unfortunately, Web services use several disparate methods for ensuring security, thereby making it difficult to incorporate security in a cache implementation.

The remainder of this paper addresses the important issues of playback, cacheability, consistency, and user feedback, but ignores hoarding and security. Hoarding and security in the context of Web services are both areas for future work. For the Web services and applications that we have studied thus far, the lack of explicit hoarding techniques has not been a problem since the set of operations is small and client access patterns have been naturally repetitious. Security has also not been a concern since we assume that the mobile devices on which the cache resides are under the control of a single user.

5 WSDL Annotations

Several of the issues faced by a Web service cache are caused by the inherent diversity of Web services. The study outlined in the previous section suggested that the Web service cache could be more effective if it had a reasonable understanding of the structure of the Web services. This section presents a generic technique to express the required semantics of a Web service. Specifically, we add annotations that describe the properties and semantics of the interface and communication protocol exposed by the Web service to the WSDL description of the Web service. The annotations extend the description of the Web service's interface and message formats. The annotations do not affect tools that automatically generate Web service clients from WSDL specifications, but are simply used to adjust the behavior of custom cache managers (as described in Section 6). Further, these annotations are optional, and a default cache behavior is defined for Web services whose WSDL documents are not annotated.

Annotating the WSDL document satisfies both our design goals of transparency and general applicability. Annotations can be added to the WSDL description without requiring any modifications to the implementation of the Web service. These annotations could either be published by the service provider or by a third party provider reasonably aware of the semantics of the Web service. The annotations use standards like XPath and XSLT and can be applied to any Web service.

Properties of Operations

A Web service's WSDL document contains an *operation* element used to specify the message format for each operation exported by the service. Our cache managers understand the following attributes that can be added to an operation element in the WSDL document.

cacheable: A *boolean* attribute that specifies whether the operation is cacheable. The default value is *false*.

lifetime: An *integral* attribute that specifies the duration for which a response for his operation should be cached. The default value is *0*.

playback: A *boolean* attribute that specifies whether the operation is an update and needs to be played back to the server upon connection restoration. The default value is *false*.

defaultResponse: A *boolean* attribute that specifies whether the cache generates a default response during disconnections. The default value is *false*. A default response needs to be sent by the cache in order to make the Web client believe that the Web service is available during disconnections when no stored response is found in the cache.

cacheHeader: A *boolean* attribute that specifies whether a cache header is appended to the response message to provide feedback to the user about operations that are serviced from the cache. The default value is *false*.

The cache manager uses the default values of these attributes when the attributes are not specified in the annotated WSDL document. The default values were chosen to provide the fewest surprises to users and applications that are not aware of the underlying cache. By default, operations are considered to be non-cacheable and are not placed in the playback queue; thus, operations whose specifications have not been annotated are simply ignored by the cache. Such operations cause the client to receive a "service unreachable" exception if performed while disconnected.

Properties of SOAP Messages

Two annotations help the cache manager understand the basic format of the SOAP messages used by a Web service. These annotations are added as attributes to the WSDL document's *binding* element, which describes the components of the SOAP messages for various operations. These attributes are expressed as strings in the XML Path Language (XPath) [2]. XPath is used to describe queries on XML documents. XPath strings can identify specific portions of the XML document as well as perform basic boolean, floating point and string operations. Thus, the XPath language is adequate to express complicated query strings. The two binding element annotations recognized by our cache manager are as follows.

operationName: An *XPath* attribute that is used to extract the name of the operation embedded in a SOAP message. The cache manager applies this XPath query to each request (XML) message to obtain the name of the operation being requested, and then uses the operation name to identify the properties of the operation from the annotated WSDL document. If this attribute is not set, then default operation properties as described earlier are assumed.

identifier: An *XPath* attribute that is used to extract the components of the request message that can be used for comparisons. Applying this query to a request message selects the set of elements of the message that are used as the key for cache lookups. In other words, these elements of two request messages can be compared to decide whether they are the same operations If this attribute is not set, the entire request message is used for comparison.

Cache Header

In order to inform the user of responses drawn from the cache during disconnected operation, a cache manager may add an optional cache header to the SOAP responses. The cache header is defined in the WSDL document like any other SOAP header and does not affect the working of existing Web clients. Further, the cache header is optional and hence can be completely ignored by unmodified Web service clients. Cache-aware Web clients can use the information provided by the cache header to apprise users of the events in the network. For example, a Web service application can read the cache header and pop-up a window to inform the user that the response was returned from the cache or that the request was stored in the playback queue to be communicated later.

The cache header, which is appended to the response message before the cache replies to the client, contains the following attributes that provide useful information about the cache and the status of network connection.

fromCache: A *boolean* attribute that indicates whether the response was retrieved form the cache.

age: An *integral* attribute that gives the age of a cached response, i.e., the duration for which the response was stored in the cache.

toPlayback: A *boolean* attribute that indicates whether the request was temporarily stored for future playback to the server.

defaultResponse: A *boolean* attribute that indicates whether the response is a default response generated by the cache manager. In that case, a cache-aware Web client may choose to ignore the response.

request 1: **query** request

```
<queryRequest select = “contact[name=‘joe’]” />
```

request 2: **delete** request

```
<deleteRequest select =
        “contact[name=‘joe’]/phone[@cat=’cell’]” />
```

Figure 3: An example for inconsistent requests from *MyContacts* Web service

Consistency Semantics

The ultimate effectiveness of the Web service cache architecture depends on how well it can support the consistency semantics of diverse Web services. Consistency requirements may demand that responses for certain requests stored in the cache be invalidated due to the execution of later requests. For example, consider the two requests of *MyContacts* Web service shown in Figure 3. The first request specifies a *query* operation asking for the retrieval of Joe’s contact information. The second request is a *delete* operation removing Joe’s cell phone number. If the response of the *query* request were stored in the cache, then the execution of the *delete* request should invalidate the cached response causing it to be removed from the cache. Expressing such consistency requirements as annotations to the WSDL document is a non-trivial task since the consistency semantics of one Web service could drastically differ from that of another service.

Consider two operations performed in succession by a Web service application: *request*$_1$ and *request*$_2$. Let the response for *request*$_1$ be currently stored in the cache. Suppose that *request*$_2$ is being currently processed. Whether *request*$_2$ invalidates *request*$_1$ depends both on the operations being performed by the two requests as well as the parameters being passed to the two operations. The condition for the invalidation of *request*$_1$ by *request*$_2$ can be expressed as a boolean function of the operation names and the parameters passed to the operations of the two requests. Thus, the consistency semantics of a Web service can be expressed in its WSDL document by defining this invalidation function.

In our current implementation, XSL transformations are used to define the invalidation function. Extensible Stylesheet Language Transformation (XSLT) [21] is an XML based standard used to convert XML documents into documents of other formats such as HTML or text. For our purposes, XSLT is a powerful procedural language with interesting features such as iterations, conditional statements, variables and function calls that can be used to express the invalidation function. Further, XSLT also accepts scripts written in other languages such as JScript, Visual Basic and C#.

The power of XSLT is used to perform smarter transformations than just specifying a boolean invalidation function. An XSLT transformation, called the *cacheTransform,* may be added as an element of the *binding* element of the WSDL document. Instead of specifying whether *request*$_2$ invalidates *request*$_1$, the XSLT transformation can actually modify the cached response for *request*$_1$ to conform to the changes requested by *request*$_2$. For example, when the delete request shown in Figure 3 is handled, a smart *cacheTransform* would actually modify the response of the earlier query request by deleting Joe’s cell phone number from the cached response.

Whenever a Web service application requests a new operation, the cache manager applies the cacheTransform to each previously cached response of this service. The cacheTransform takes four parameters as input: some *request*$_1$ whose response is stored in the cache, *response*$_1$ stored in the cache, *request*$_2$ that is being currently processed and *response*$_2$ of *request*$_2$ if the Web service is currently available. The *cacheTransform* transforms the *response*$_1$ according to the semantics of the Web service if possible. If *request*$_2$ does not invalidate *request*$_1$, then the transform produces the original *response*$_1$ as its output. If *request*$_2$ invalidates *request*$_1$ but *response*$_1$ cannot be suitably modified, the transform outputs an empty string signifying that *response*$_1$ must be removed from the cache. When no *cacheTransform* is specified in the WSDL document, a default XSL transform that always outputs *response*$_1$ (does not invalidate) is applied.

The cacheTransform provides enormous flexibility to the users for choosing the level of consistency they want. The level of consistency is defined by the cacheTransform and not stipulated by the Web service cache. For example, a cacheTransform guaranteeing stronger consistency would perform more invalidation while a cacheTransform guaranteeing weaker consistency might modify the original response. Multiple WSDL annotations for the same Web service can be made available each specifying different levels of consistency. The users have the freedom to determine what kind of inconsistency they are willing to tolerate.

Other XSL Transforms

We define two other XSL transforms namely, *defaultResponseTransform* and *responseTrasform* to complete the basic functionality of a Web service cache. The *defaultResponseTransform* is used to generate the default response for an update operation during disconnections. This transform takes the currently handled request as the input and produces a default response that can be sent to the Web client. If this transform is not specified in the WSDL document, a default transform that always produces an empty string is used instead.

The *responseTransform* is applied on a response retrieved from the cache to modify it suitably to be returned to the client. For example, the response message may need to contain a field stating the time of processing the request. The *responseTransform* is used to impart such modifications to the response message retrieved from the cache. The responseTransform takes as input a response message and outputs the modified version of the message. If this transform is not defined in the annotated WSDL document, a default transform that outputs the unmodified response is applied.

These WSDL annotations permit the Web service cache manager to provide acceptable consistency during disconnections and improve the availability of Web services. Section 7 discusses the actual annotations used for the *MyContacts* Web service. Before getting into that level of detail, however, the next section explains the architecture and role of custom cache managers.

6 Web Service Cache Architecture

An adaptive Web services cache architecture has been developed that adjusts its behavior for different Web services based on the WSDL annotations to improve availability of Web services. A graphical illustration of our architecture is shown in Figure 4.

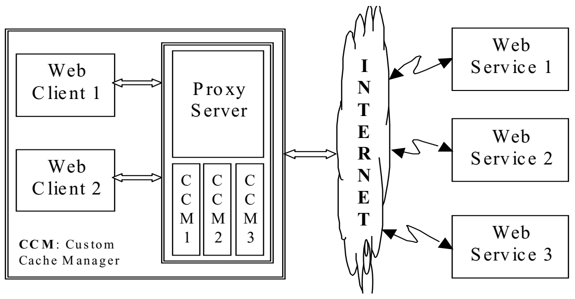

Figure 4: Web Service Cache Architecture

The Web service cache resides in a HTTP proxy server on the client device. All HTTP request messages from and to Web clients pass through this proxy server. Deploying the Web service cache in this manner provides transparency. The proxy server delegates all the Web service requests (SOAP messages) to the Web service cache and acts merely as a tunnel for all other HTTP messages. The proxy server recognizes a SOAP request by the presence of the *soap-action* header in the HTTP request message. The Web service cache consists of three components, which are described below.

Cache Store

The cache store is responsible for storing and retrieving Web service requests and responses. A numeric key that is generated by the custom cache manager identifies each cache entry. A pre-specified amount of space is allocated on the file system and the Web requests and responses are stored as files in disk storage. Each cache entry is associated with a finite lifetime as dictated by the cache manager. Entries from the cache are deleted permanently upon expiry. The cache store also implements the LRU (Least Recently Used) strategy for cache replacement when the cache is full.

Write Back Queue

The write back queue is responsible for periodically checking the network for connectivity and playing back queued up requests to the Web service provider. Web service requests that could not be handled due to network outage at the time of reception are stored in the write back queue. Queued Web requests are stored as files in the machine's local disk. Periodically, the write back queue checks for network connectivity by issuing the first request in the queue. If the network is still disconnected, the write back queue waits for a certain duration and repeats the same process again. If network connectivity is perceived, the queued Web service requests are issued in first-in-first-out order. The responses to these requests are cached, as in normal operation, possibly replacing outdated cache entries.

Custom Cache Manager

A custom cache manager is generated automatically for each Web service that has an annotated WSDL specification. The custom cache manager controls the behavior of the cache according to the annotations described in the WSDL document. The Web service cache detects the name of the Web service from the request URI (Uniform Resource Identifier) and delegates the request message from the Web client to the appropriate custom cache manager. The custom cache manager handles the Web service request according to the properties described in the annotated WSDL document of that Web service.

The custom cache manager applies the XPath query defined by the *operationName* attribute in the WSDL document to the request message in order to obtain the name of the operation being invoked. Next, it applies

```
<binding name="myContactsBinding" type="tns:myContactsPort"
  operationName = "substring-before(localname(/e:Envelope/e:Body/*[1]), 'Request')"
  identifier = "/e:Envelope/e:Header/s0:licenses | /e:Envelope/e:Header/s1:request | /e:Envelope/e:Body">

    <s:binding transport="http://schemas.xmls.org/s/http" style="document" />

    <operation name="insert" cacheable="false" playback="true" defaultResponse="true" cacheHeader="true">
```

Figure 5: Sample WSDL annotations for *MyContacts* Web service.

the XPath query defined by the *identifier* attribute in the WSDL document to extract portions of the request message useful for identification. The extracted parts of the request message are hashed to obtain a numeric key that identifies this request in the cache store.

The custom cache manager probes the network for connectivity. A network disconnection is assumed if either the write back queue is not empty or if the Web server does not reply within a time out period. If the network is connected, the service request is issued to the server and the response obtained. If the operation is cacheable (detected by reading the WSDL annotations), the new response is stored in the cache replacing any old response for this request.

The custom cache manager performs the following activities when the network is disconnected. If the operation is an update the request is stored in the write back queue for future replay. If the operation is cacheable and the cache store has a valid entry for this request, the response is obtained from the cache and the *responseTransform* specified in the WSDL document (or the default version if none is specified) is applied to it. The transformed response is then sent to the Web client after adding the appropriate cache header if stipulated in the WSDL document. Otherwise, a default response is generated by applying the *defaultResponseTransform* specified in the WSDL document to the request message. This default response is appended with an appropriate cache header if required and sent to the Web client. If the *defaultResponseTransform* is not provided, no response is sent to the Web client.

The custom cache manager also applies the *cacheTransform* specified in the WSDL document (or the default version). This transform is applied to each entry in the cache store that belongs to this Web service. The Web request and its cached response along with the new request are passed as parameters to the XSL transform. The new response obtained from the server is also passed to the cacheTransform if the client is not disconnected. The transformed response output by the *cacheTransform* is then stored in the cache. If the *cacheTransform* produces an empty string as output, the cache entry is deleted from the cache store. Applying the *cacheTransform* to each entry in the cache is expensive. Nevertheless, consistency checks are obligatory for every operation. Performance degradation due to consistency enforcement could be limited by lazy (delayed) application of the *cacheTransform* after sending the response to the Web client. However, care must be taken to ensure that all pending transforms are applied before servicing other requests.

Default Cache Manager

If the proxy server receives a request destined for a Web service for which no annotated WSDL document is available, then the default cache manager handles the request. In general, no single default caching behavior works best for all Web services, applications, and users. For regularly used Web services, annotating the services' WSDL documents is recommended so that custom cache managers can be used. The default cache manager, however, can be useful in those cases where annotations are not available. Users can configure the defaults used by the cache manager.

For users wishing to maximize cache-hit rates, that is, maximize the availability of data while disconnected, having the default cache manager treat all requests as cacheable is desired. This is essentially what was done for the study reported in Section 4. Caching everything works well for the large class of Web services that only provide query operations on fairly static data, such as map services, currency conversion services, yellow page services, and so on.

Setting playback as the default option for operations ensures that update operations are eventually performed when connectivity is restored. However, this incurs the cost of also queuing and issuing non-update operations. Moreover, it requires that a default response be generated for each request that results in a cache miss. The default cache manager can generate a properly formatted response since the structure is specified in the service's WSDL document, but the default values supplied for each field may be meaningless to the application.

The conservative approach is to configure the default cache manager so that operations are non-cacheable

```
<?xml version="1.0" encoding="utf-16"?>
<s:Envelope xmlns:s=http://schemas.xmlsoap.org/soap/envelope/ xmlns:hs="http://schemas.microsoft.com/hs/2001/10/core">
  <s:Header>
      <path xmlns="http://schemas.xmlsoap.org/rp/">
          <action>http://schemas.microsoft.com/hs/2001/10/core#response</action>
          <rev></rev>
          <from>http://microsoft-m3we4f.microsoft.com</from>
          <relatesTo > d978b559-aceb-4e9e-9747-b8a306234bc8 <relatesTo>
      </path>
      < response xmlns ="http://schemas.microsoft.com/hs/2001/10/core" />
      <cacheHeader defaultResponse="true" toPlayback="true" xmlns="http://localhost/wsdlannotation" />
  </s:Header>
  <s:Body>
      <hs:insertResponse status="success" selectedNodeCount="1" newChangeNumber="0" />
  </s:Body>
</s:Envelope>
```

Figure 6: Sample default response message for *MyContacts* Web service.

and are not placed in the playback queue. These are the same defaults used when an annotated WSDL document is available but does not contain annotations for some operations. In this case, the default cache manager does nothing but act as a tunnel for requests and responses. These operations are thus unavailable when the client is disconnected.

Implementation

We have implemented a prototype of this architecture in Microsoft's .NET framework using the C# programming language. We built a proxy server that complies with the HTTP 1.1 protocol standard [12]. The Web service cache as described in this section has been built and incorporated with the proxy server. The proxy server delegates all SOAP messages to the Web service cache and serves as a tunnel for all other HTTP requests. The write back queue probes for network connectivity once every 2 minutes.

We have tested this prototype against the .NET My Services by artificially invoking network disconnections. The results of this deployment have been encouraging. Specifically, we were able to demonstrate that the .NET My Services set of Web services such as *MyContacts* can be used during disconnections with little awareness of the disconnection. In the next section, we describe the annotations made to the WSDL document of the *MyContacts* Web service used to test our prototype. We plan to repeat this study with additional Web services as they become publicly available in order to further validate the viability of our cache architecture.

7 Case Study: *MyContacts* Web Service

The *MyContacts* Web service belongs to the .NET My Services. This Web service allows users to store contact information of friends and acquaintances. This Web service exports four significant operations, namely *query*, *insert*, *delete* and *replace*. The *query* operation lets users select portions of the contact information. This operation takes an XPath string as the query expression. The *insert* operation takes contact data to be inserted in the database as well as an XPath string specifying the location for the insertion. The *delete* operation takes an XPath query string and deletes the specified entries in the database. The *replace* operation performs an atomic delete and insert in the database. For a detailed description of the .NET My Services Web services see the .NET My Services Specification [27].

Figure 5 shows parts of the annotated WSDL document for the *MyContacts* Web service. The annotations added to the WSDL document are shown in bold. There are two attributes added to the *binding* element. The *operationName* attribute specifies an XPath expression that extracts the name of the operation from the request message. In this case, XPath expression extracts the name of the first element contained in the body of the soap envelope and gets the operation name from it. Looking back at Figure 1, which shows a sample request message for the *insert* operation of *MyContacts* Web service, note that the name of the first element in the body of the SOAP envelope is *insertRequest* and the operation name can be obtained by removing the word *Request* from it.

The *identifier* attribute in the *binding* element specifies an XPath expression that selects the *licenses* header, the *request* header and the *Body* elements from the request message. The sample insert request in Figure 1 contains the *path* header in addition to these headers. The *path* header contains the id element that specifies a unique identifier for this message. The identifier attribute in the annotated document extracts all portions of the message except the *path* header for identification.

```
<xsl:template match="/">
   <xsl:variable name="service1" select="$req1/s:Header/c:request/@service"/>
   <xsl:variable name="service2" select="$req2/s:Header/c:request/@service"/>
   <xsl:variable name="opName1" select="substring-before(local-name($req1/s:Body/*[1]), 'Request')"/>
   <xsl:variable name="opName2" select="substring-before(local-name($req2/s:Body/*[1]), 'Request')"/>
   <xsl:choose>
      <xsl:when test="$service1 = $service2">
         <xsl:choose>
            <xsl:when test="$opName2 = 'query' and ($opName1 = 'insert' or $opName1 = 'delete' or $opName1 = 'replace')">
               <xsl:variable name="cleanQuery1">
              <xsl:call-template name="StripSegment">
                 <xsl:with-param name="xpQuery" select="substring-after($req1/s:Body/c:*/@select, '/')"/>
              </xsl:call-template>
               </xsl:variable>
               <xsl:variable name="cleanQuery2">
              <xsl:call-template name="StripSegment">
                 <xsl:with-param name="xpQuery" select="substring-after($req2/s:Body/c:queryRequest/c:xpQuery/@select, '/')"/>
              </xsl:call-template>
               </xsl:variable>
               <xsl:call-template name="CheckIntersection">
              <xsl:with-param name="xpQuery1" select="$cleanQuery1"/>
              <xsl:with-param name="xpQuery2" select="$cleanQuery2"/>
               </xsl:call-template>
            </xsl:when>
            <xsl:otherwise>
               <xsl:value-of select="$rep2"/>
            </xsl:otherwise>
         </xsl:choose>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="$rep2"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Figure 7: Sample invalidation cacheTransform for *MyContacts* Web service.

Figure 5 also shows the properties of the *insert* operation that have been added to the operation element of the WSDL document. The *insert* operation is defined as an update but not cacheable. These attributes also specify that a default response should be generated when the network is disconnected and that a cache header should be appended to the messages sent by the Web service cache. The response messages of this Web service typically have a status code indicating success or failure. The *defaultResponseTransform* for this Web service would just generate a skeleton response indicating success of the specified operation. Figure 6 shows a sample default response for the *insert* operation. The figure also shows the cache header appended to the default response. Attributes in the cache header inform the Web client that the request was stored for future play back and that the current response is a default response generated by the Web service cache. A cache aware Web client would interpret the cache header and report this information to the user.

Every response message of the *MyContacts* Web service has a field called *relatesTo* (see Figure 6) in the path header. The value of this field refers to the unique identifier of the request message that generated this response. The *responseTransform* for this Web service copies the value of the unique identifier from the request message to the *relatesTo* field of the path header.

An *insert*, *delete* or *replace* operation could invalidate a *query* operation. Whether the current operation invalidates a past operation whose response is stored in the cache depends on whether the two XPath select strings intersect. By intersection of two XPath expressions we mean overlap in the location specified by the two strings. For example in Figure 3, both the query strings point to the information of the same contact (Joe) and so the select strings intersect. Figure 7 shows a *cacheTransform* that specifies invalidations based on this principle of intersection. This transform ignores the predicates (sub-expressions in the square brackets) in the select expressions and compares the location path specified in the two select strings to determine intersection. Two location paths intersect if they are the same or if one is a prefix of the other. This transform provides a strong form of consistency for a user's own operations.

The *cacheTransform* shown in Figure 7 only specifies invalidations but does not modify the response in the cache. In our trial deployment, we employed a smarter *cacheTransform* that suitably modifies the responses in the cache. The smarter transform locates the path specified by the insert, delete and replace request in the query response stored in the cache, performs the respective operation and modifies the query response. If the operation could not be performed on the query response (due to uncertainties in the location path), it resorts to specifying invalidations as described before. The smart *cacheTransform* and cache header made a significant difference to the user experience and satisfaction during our trial deployment. Users were able to invoke a number of operations during disconnections and yet observe the operations being effected on the cached responses while being warned by the cache aware Web client.

8 Conclusions and Future Work

In this paper, we have described an XML Web service cache architecture. While cache managers could be used to improve performance, our focus has been on improving the availability of Web services during network disconnections. Our cache managers can be deployed transparently between client applications and servers without alterations to the Web service implementations and communication protocols. We have annotated WSDL descriptions of Web services in order to be able to efficiently support diverse XML Web services. We built a prototype implementation and tested it for .NET My Services such as *MyProfile*, *MyContacts*, and *MyFavoriteWebSites* by invoking artificial disconnections.

The main contributions of our work thus far are (1) identifying the practical issues that arise in caching non-trivial Web services and (2) demonstrating through an experiment that request/response caching can provide satisfactory support and experience to mobile users for one class of existing Web services. However, the overall utility of our caching architecture for disconnected operation remains to be fully evaluated. As new mission-critical Web services emerge over the next few years, we plan to explore the range of services for which a caching scheme of this sort can be effective.

Areas for future work include further improving the availability of the Web services cache by hoarding. Providing user controlled means to specify hoard requests, smart cache transforms operating on cached responses of hoard requests and the ability to respond to requests with the help of cached responses are important steps to pursue. Also we have currently ignored a number of issues related to security in this architecture. Adding support for handling security in the Web service cache would be another important goal. Finally, an interesting challenge exists in developing tools that automatically deduce the properties of operations exported by Web services for which annotated WSDL specifications are not available. The open issue is whether a cache manager can reasonably figure out the characteristics of a Web service from passively observing requests and responses thereby improving the default caching behavior.

Bibliography

[1] Daniel Barbará and Tomasz Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments. *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, May 24-27, 1994, pages 1-12.

[2] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, Jérôme Siméon. W3C Working Draft "XML Path Language (XPath) 2.0", 16 August 2002. (See http://www.w3.org/TR/xpath20/.)

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. W3C Recommendation "Extensible Markup Language (XML) 1.0 (Second Edition)", 6 October 2000. (See http://www.w3.org/ TR/2000/REC-xml-20001006.)

[4] P. Cauldwell, et. al. *Professional XML Web Services*. Wrox Press Ltd. Birmingham, U. K. 2001.

[5] Boris Y. L. Chan, Antonio Si, Hong Va Leong: Cache Management for Mobile Databases: Design and Evaluation. *Proceedings of the Fourteenth International Conference on Data Engineering*, February 23-27, 1998, Orlando, Florida: pages 54-63.

[6] Bharat Chandra, Mike Dahlin, Lei Gao, Amjad-Ali Khoja, Amol Nayate, Asim Razzaq, and Anil Sewani. Resource management for scalable disconnected access to Web services. *Proceedings of the Tenth International Conference on World Wide Web*, 2001, Hong Kong, Pages: 245 – 256.

[7] Henry Chang, Carl Tait, Norman Cohen, Moshe Shapiro, Steve Mastrianni, Rick Floyd, Barron Housel, and David Lindquist. Web Browsing in a Wireless Environment: Disconnected and Aynschronous Operation in ARTour Web Express. *Proceedings MobiCom*, 1997, Budapest, Hungary, pages 260-269.

[8] Ing-Ray Chen, Ngoc Anh Phan, and I-Ling Yen. Algorithms for Supporting Disconnected Write Operations for Wireless Web Access in Mobile Client-Server Environments. *IEEE Transactions on Mobile Computing*, Vol. 1, No. 1, January-March 2002, pages 46-58.

[9] Roberto Chinnici, Martin Gudgin, Jean-Jacques Moreau, Sanjiva Weerawarana. W3C Working Draft "Web Services Description Language (WSDL) Version 1.2", 9 July 2002 (See http://www.w3.org/TR/wsdl12/.)

[10] Gregory V. Chockler, Danny Dolev, Roy Friedman, and Roman Vitenberg. Implementing a Caching Service for Distributed CORBA Objects. *Proceedings International Conference on Distributed Systems Platforms (Middleware)*, New York, New York, 2000, pages 1-23.

[11] Curbera, F. Duftler, M. Khalaf, R. Nagy, W. Mukhi, N., and Weerawarana, S. Unraveling the Web services Web: an introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, March-April 2002, Volume: 6 Issue: 2, pages 86 – 93.

[12] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk Nielsen, T. Berners-Lee. IETF "RFC 2616: Hypertext Transfer Protocol - HTTP/1.1", January 1997. (See http://www.ietf.org/ rfc/rfc2616.txt.)

[13] Rick Floyd, Barron Housel, and Carl Tait. Mobile Web Access using eNetwork Web Express. *IEEE Personal Communications*, Vol. 5, No. 5, October 1998, pages 47-52.

[14] Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen. W3C Working Draft "SOAP 1.2 Part 1: Messaging Framework", 2 October 2001 (See http://www.w3.org/ TR/soap12-part1.)

[15] Martin Gudgin, Marc Hadley, Jean-Jacques Moreau, Henrik Frystyk Nielsen. W3C Working Draft "SOAP 1.2 Part 2: Adjuncts", 2 October 2001 (See http://www.w3.org/TR/soap12-part2.)

[16] R.G. Guy, J.S. Heidemann, W. Mak, T.W. Page, Jr., G.J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *Proceedings Summer USENIX Conference*, June 1990, pages 63-71.

[17] Joanne Holliday, Divyakant Agrawal, and Amr El Abbadi. Disconnection Modes for Mobile Databases. *Wireless Networks*, Issue 8, 2002, pages 391-402.

[18] Jin Jing, Abdelsalam Sumi Helal, and Ahmed Elmagarmid. Client-server computing in mobile environments. *ACM Computing Surveys* Volume 31 , Issue 2 (June 1999), Pages: 117 – 157.

[19] Anthony D. Joseph, M. Frans Kaashoek, Building reliable mobile-aware applications using the Rover toolkit, *Wireless Networks*, v.3 n.5, p.405-419, Oct. 1997.

[20] Anupam Joshi. On proxy agents, mobility, and Web access. *Mobile Networks and Applications*, Volume 5, Issue 4 (December 2000), Pages: 233 – 241.

[21] Michael Kay. W3C Working Draft "XSL Transformations (XSLT) Version 2.0", 16 August 2002. (See http://www.w3.org/TR/xslt20/.)

[22] Kahol, S. Khurana, S.K.S. Gupta and P.K. Srimani. A strategy to manage cache consistency in a disconnected distributed environment. *IEEE Trans. on Parallel and Distributed Systems*, Vol. 12. No. 7, July 2001, pp. 686-700.

[23] James J. Kistler, M. Satyanarayanan, Disconnected operation in the Coda File System, *ACM Transactions on Computer Systems (TOCS)*, v.10 n.1, p.3-25, Feb. 1992.

[24] R. Kordale, M. Ahamad, and M. Devarakonda. Object Caching in a CORBA Compliant System. *Proceedings USENIX Conference on Object-Oriented Technologies (COOTS)*, Toronto, Canada, June 1996.

[25] Geoffrey H. Kuenning and Gerald J. Popek. Automated hoarding for mobile computers. *Proceedings of the sixteenth ACM symposium on Operating systems principles*, Saint Malo, France, 1997, pages 264 – 275.

[26] Barbara Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shira. Safe and Efficient Sharing of Persistent Objects in Thor. *Proceedings International Conference on Management of Data (SIGMOD)*, 1996, Montreal, Quebec, Canada, pages 318-329.

[27] *Microsoft .NET My Services Specification*. Microsoft Press. Redmond, Washington. 2001.

[28] Michael N. Nelson and Yousef A. Khalidi. Generic Support for Caching and Disconnected Operation. *Proceedings Fourth Workshop on Workstation Operating Systems*, Napa, CA, October 1993, pages 61-65.

[29] Mark Nottingham. SOAP Optimization Modules: Response Caching. W3C Draft, August 2001. http://lists.w3.org/Archives/Public/www-ws/2001Aug/ att-0000/01-ResponseCache.html.

[30] Mark Nottingham. Optimizing Web Services with Intermediaries. *Proceedings of the Sixth International Workshop on Web Caching and Content Distribution*, Boston University, Boston, Massachusetts, USA, June 20-22, 2001.

[31] Matt Powell. XML Web Service Caching Strategies. Microsoft Corporation, MSDN Library, April 17, 2002. http://msdn.microsoft.com/library/default.asp?url= /library/en-us/dnservice/html/service04172002.asp.

[32] D. B. Terry, M. M. Theimer , Karin Petersen , A. J. Demers , M. J. Spreitzer , C. H. Hauser, Managing update conflicts in Bayou, a weakly connected replicated storage system, *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, Copper Mountain, Colorado, December 1995, p.172-182.