Transparent Caching of Web Services for Mobile Devices

Kamal Elbashir
Multi-Agent Distributed Mobile and
Ubiquitous Computing Laboratory
Computer Science Department
University of Saskatchewan
(306) 966-4744
kae501@cs.usask.ca

ABSTRACT

A Web service is a collection of functions packaged, published and consumed using standard Internet protocols. This paper presents a generally applicable architecture for transparent Web Service caching, with a focus on Mobile Devices. This approach defines a set of required Semantics embodied in a document called the ‘Service Semantics Description’ (SSD). The SSD is used by the Client-side cache and (optionally) by the intermediate Caching Proxy. A tool facilitating definition of semantic tags is developed as an IDE extension. Preliminary results are presented outlining the feasibility, transparency, general applicability, and initial measurements of CPU and Memory overheads.

1. INTRODUCTION

Consuming a Web Service is accomplished by sending service requests (method calls), the service responds with a method return. Requests and responses are couriered as SOAP (Figure 1) messages (Simple Object Access Protocol). Accompanying a Web Service is the service’s WSDL (Web Service Description Language), specifying syntax metadata. An optional layer, the Universal Description and Discovery Interface (UDDI) offers a directory service for discovery of web services [3, 4, 7, 12].

Communication with a Web Service is established using standard Internet protocols (e.g. HTTP). HTTP is the protocol underlying the World Wide Web. Adoption of Web Services gains remarkably from the utilization of well established protocols of the heterogeneous network of the WWW [4, 12].

A Web Service client must obtain the service’s WSDL document. The WSDL document contains syntactic information about the service (Namespaces, Method signatures, Data types and a URI). A client consumes a web service by initiating the exchange of SOAP messages (see Figure 1-A). The messages must follow the inscriptions detailed in the WSDL document (URI, Signatures, and Data Type information) [4, 12].

For a mobile device, consuming a web service is an attractive interoperable approach to access remote business logic. Mobile devices are constrained by Battery, CPU, Memory and Weak Connectivity. Weak Connectivity results from intermittent communication due to low bandwidth, high latency or expensive networks. Furthermore, Mobile devices are susceptible to both voluntary and involuntary disconnections (user initiated disconnections vs. disconnections due to a change in the availability of a resource, e.g. Battery life). For the mobile user, constraining productivity to times of full connectivity hinders a successful and usable deployment of remote business logic [1, 7].

This paper will refer to a mobile device consuming a Web Service (a Web Service Client) as a Mobile Host (MH). A Cache is a temporal memory coordinated by a Caching Policy, cache records are memorized instances of data (Requests\Responses). A Proxy is an entity acting as an intermediate connectivity tunnel, a proxy maybe caching (Caching Proxy) [6]. A caching proxy plays an intelligent role while tunnelling requests to their respective endpoints. Cacheability of a request (method call\return cacheability) is the property stating that the request maybe cached while maintaining application logic (no negative side effects on the application logic) [3, 8].

2. PROBLEM DEFINITION

2.1 Introduction

Operation of a mobile service client is restricted to times of connectivity. The service Provider may suffer network or system downtimes. Furthermore, the mobile device is constrained by Weak network connectivity. When the mobile client is disconnected (null connectivity) the service is inaccessible, and the mobile application is unusable. Null connectivity occurs when the device is out of network range, or when the device is voluntarily or involuntarily disconnected (e.g. user-initiated power-off vs. a depleted battery state). Additionally, network infrastructures supporting mobility are bandwidth-limited. Clients consuming rich services suffer degradation of response times due to latencies incurred when sending large upstream requests (service arguments) and receiving of large responses (return values).

This paper will explore the issues present when a MH loses network connectivity (null connectivity) and the necessary recovery logic upon reconnection (reintegration phase). The user experience of a mobile device is reflected by the user’s inability to continue working while the MH is disconnected. An improvement of the user experience can be achieved by enabling the user to continue to work while in disconnected mode. Increased application response times, due to caching of frequent requests and\or Prefetching of anticipated future requests is a desirable improvement.

The mobile application is assumed to be resilient to stale resource access (invalid, out of age resources). This implies that the application logic is not broken when a request is answered from the cache. This is a necessary assumption since the consistency of a cached resource cannot be guaranteed while operating in null connectivity.

2.1.1 *Web Service Caching*
A Web Service, described by the WSDL document can be viewed as a Remote Object, accessible by SOAP messaging over HTTP. Methods and properties exposed by the remote object are accessed using SOAP messages adhering to the calling convention prescribed by the WSDL. Every SOAP request\response corresponds to a single method invocation or a single property set\get operation [3, 4, 7, 12].

A naive caching architecture stores tuples of request\response pairs. When a new request is made; and a matching request is in the cache, then the corresponding response is returned (cache hit). If a similar request is not in the cache (a cache-miss); and the MH is operating in disconnected mode; then an exception is raised. All attempted requests, while in disconnected mode, are stored in a 'Pending' FIFO queue for execution when connectivity is restored (Reintegration Phase). Upon reconnection, the reintegration phase commences by issuing pending requests, an attempt of synchronizing the state of the disconnected cache with the state of the remote object.

The relationship between method calls and the state of the service is crucial. Service methods fall into one of three types: Read-methods (state-reading) and Write-methods (state-altering), or State-independent methods. Properties offered by the service are inherently state-reading and state-altering [3, 8]. The following discussion is only concerned with service methods; the outcome can be generalized to service Properties.

The classification of a method as state-reading, state-altering, or state-independent is best known by the publisher of the Web Service, since services implementations are exposed as black-boxes. It is impractical to assume that all methods are of one type or another (in reference to their state dependency). There is no standard for specifying Semantic information about Web Service methods. The WSDL document is limited to describing only the Syntactic information relating to service consumption [3, 8, 12].

A Web Service caching architecture, built specifically for Mobile devices must consider the limited processing and space constraints available to the mobile device (MH). These constraints limit the allowable Cache-size, and the processing requirements of cache management and operation [1, 3]. The architecture should preserve the logic consistency of the mobile application, a mobile application functioning with and without a cache should experience no side effects resulting in incorrect operation. Out of age cache records should be invalidated. Cross-MH cache consistency is a requirement for mission critical mobile applications (e.g. shop floor applications) [1, 3, 8].

Caching in the area of the World Wide Web is focussed on Read-caching [5, 6]. Read\Write caching has been well explored in the areas of File systems, Databases and Distributed Object Systems [2]. Many issues faced by a mobile Web Service client are comparable to aspects of caching in the aforementioned paradigms. This paper explores previous research in section 3.

Approaches common to Web caching are severely limited when applied to caching of Web Services. Static web pages need only Read-caching. Only when a sophisticated web caching system is utilized (e.g. Active Caching, for dynamic web content) then a form of Write-caching is present [5, 6]. A modified web caching proxy may treat the Web Service as a dynamic page and cache the SOAP messages that are couriered in the body of HTTP requests. Such architecture is very limiting and its assumptions are unrealistic. The modified caching proxy must compare bodies of SOAP messages when testing for a cache-hit, the message body may include metadata that is not necessarily part of the method call (e.g. RequestID), resulting in false cache-misses [3]. Another problem with such an implementation is the proxy's inability to replay state-altering requests upon restoration of service availability. Finally and most importantly, the meaning and structure of a web page is fundamentally different from that of a web service, the required semantic information enabling caching is hardly translatable into the domain of web caching.

2.1.2 *Mobility and User Experience*
A cache for mobile devices must support transparent switching between connected mode (remote-execution) and null connectivity mode (cache-based execution).

Furthermore, service methods requiring large arguments incur delays due to network transfer latencies. Similar latencies are incurred when methods return large dataset are executed. Repeated duplicate requests should only be returned from cache and bandwidth utilization should be minimized when possible [7, 8].

2.1.3 *Caching-Architecture Deployment and Existing Web Services*
A caching architecture may require the deployment of specialized components on the server-side, a difficult requirement in real-world scenarios, service Providers are generally reluctant to alter existing implementations. An approach to transparency is achievable by caching independently of the service implementation, utilizing a Caching Proxy [7, 8]. Caching schemes employing an intermediate proxy are extensively used for caching in the areas of the WWW and Distributed Object Systems. An intermediate proxy implementation appears to be the service client when viewed by the service provider. When viewed from the client's perspective, the proxy appears as the original Web Service.

2.1.4 *Cache Location*
A client-size cache (see figure 1-B) permits the client application to recover from service requests when the service is unavailable (network outage or service downtime). If the client is caching independently of the service, unnecessary executions on the server if the client issues an expensive request and network connectivity is lost before a service response is received. Furthermore, notifications of invalidated client-cache records are harder to implement since the server is unaware of the client cache. Finally, multiple caching clients are not centrally coordinated, the clients maybe within range to form an Ad-hoc network and exchange cache-hits while the service is unavailable.

A second cache (see figure 1-C), residing at an intermediate layer between the mobile device and the web service is a promising approach. The intermediate cache (caching proxy) must not be susceptible to network disconnections (see Figure 1). Such an organization permits proxy-led coordination of client caches. The service is now shielded from processing duplicate requests made by multiple mobile devices. A proprietary protocol, optimizing the link between the intermediary proxy and the mobile device is now possible (e.g. offering compressed streams). Invalidation reports are more readily deliverable to mobile clients, as the intermediate caching proxy becomes the centralized cache coordination layer.

2.2 Key Questions

From the previous discussion, several questions arise:

1. What metadata is necessary to enable caching of Web Services? How can I enable a developer to specify the necessary metadata with minimal effort?
2. What should be cached?
3. How can a cache that is independent of service implementations be implemented?
4. What consistency guarantees are attainable?
5. What prompts cache invalidation? What happens in the reintegration phase?
6. How does existing research help in designing a practical caching architecture for Web Services?

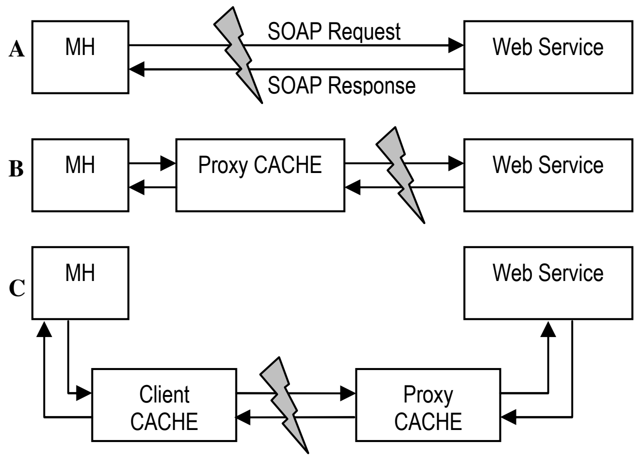


Figure 1: MH - Web Service Connectivity w\out a Cache

3. RELATED WORK

3.1 Web Caching: A Survey of Web Caching Schemes for the Internet

Wang's paper [5] is a survey of caching architectures on the Web. Proxy servers are recognized as an effective solution for bandwidth optimization, availability, and scalability. Web Proxy servers sit between the web client and the web server, mimicking the behavior of the real server. Caching may occur on the server, the proxy or the client. A set of significant improvements are recognized as following [5]:

1. Web caching optimizes bandwidth usage by reducing network traffic as more and more resources are found in the cache.

2. Web caching reduces access latency due to the fact that documents maybe found in the local cache (on the client) or on nearby proxy servers, reducing required network traffic when fetching documents. Because of the traffic reduction, more bandwidth is available for a cache-miss to be retrieved quickly.

3. The web server's performance is improved as more requests do not reach the server and documents are retrieved from the client or proxy caches.

4. Improved robustness, document availability is significantly improved due to replicas existing in local and remote caches. If the server goes offline; cached documents are still available to interested clients.

5. Proxy servers act as intermediary traffic deltas, data collection and statistical analysis of access patterns is more easily doable.

6. Load balancing is in effect, as proxy servers reduce the server's utilization by forwarding requests to the least utilized server.

Wang recognizes disadvantages of web caching, most importantly is cache consistency maintenance, as stale records maybe served to clients while the server is offline. Increased request overhead, due to processing by intermediary proxy (or proxies). A proxy is also recognized as a point of failure, the proxy's availability becomes more critical than the web server's availability. This is because one proxy server may act as a caching agent (delta) for multiple web servers [5].

Wang outlines ten desirable properties of a web caching architecture, namely: Fast Access, Robustness, Transparency, Scalability, Efficiency, Adpativity, Stability, Load Balancing, Ability to deal with Heterogeneity and Simplicity. A number of caching architectures are analyzed including Hierarchical, Distributed and Hybrid. Hierarchical caching producing the shortest connection latencies when compared to distributed caching. On the other hand, distributed caching is found to have the shortest retrieval latency but with greater bandwidth utilization [5].

Two common approaches to cache routing and optimization are recognized, by growing a caching distribution tree formed by nodes of intermediary proxy servers. Cache resolution is performed by a routing table or a hash function [5].

Prefetching [9] is needed in order to maximize the cache-hit rate, this is done by anticipating future requests and prefetching the corresponding documents pre-demand. Prefetching maybe executed between the client and the web server, between the client and the proxy agent, or between the proxy agent and the web server. Caching between the client and the web server is recognized for increasing the cache hit-rate by 45% at the cost of doubling network traffic. Rate-controlled prefetching is recognized as a possible solution to the unacceptable increase of network traffic [5].

Prefetching between the proxy agent and the web server provides significant improvements over the previous approach offering a very good predictability of client access patterns at the proxy [5]. Furthermore, due to the decreased network traffic downstream to the client. Prefetching between clients and proxy agents may best support caching for mobility, as the client and the proxy agent cooperate to cache and prefetch documents without the server's intervention [5].

A number of cache placement and replacement strategies are identified and analyzed in regards to cache size\speed. A good cache placement\replacement strategy is very important when caching for mobility. Two types of cache coherencies are recognized, Strong and Weak. Strong cache coherency is maintained either by client validation or Server invalidation. The former results in higher cache-hit latencies, while the latter requires server cooperation. Server cooperation maybe needed for broadcasting invalidation reports. Weak coherency is the case when cache records are invalidated by a timeout (TTL value). Piggyback invalidation requires the server cooperation, by sending lists of invalidated items attached to responses of new requests. The requirement for cache coherency depends largely on the client's tolerance for stale resources and on the frequency of resource changes [5].

Proxy placement is very critical for optimal performance gains. The desired properties inherit many of the desired properties of a web caching system [5].

Caching of Dynamic web resources, a topic of interest when considering caching of web services is briefly touched upon. Read-caching is the focus of the majority of web caching strategies and studies [5]. The two widely used approaches to dynamic-resource caching are identified as Active Caching and Server Accelerators. The former requires computation applets attached to dynamic portions of a page to be sent to the proxy and the latter is done by exposing a set of caching APIs at the accelerator entity, the dynamic resources at the server must utilize the APIs in order to achieve caching of dynamic resources [5]. Both approaches violate at least two of the desired properties outlined at the beginning of the paper, namely the ability to deal with heterogeneity and simplicity.

3.2 Web Service Caching for Mobility

Terry et al. [3] discuss the need for caching of Web Services to support mobility. The motivation is to offer disconnected operation of web services. Transparent deployability and General applicability are two desired properties of a web service cache. The paper distinguishes attributes specific to caching of web services, when compared to caching in the areas of file systems, and databases as both offering standard interfaces with well known semantics (Read\Write and Query\Insert\Update\Delete), and Web caching is limited to read caching [3].

The authors use the .NET MyServices set of services for an experiment in caching for disconnected operation. .NET MyServices are services offering personal profile maintenance and a contacts directory. The exposed operations are Query, Insert, Update and Delete operations [3].

The proposed architecture utilizes an intermediate SOAP proxy agent, caching SOAP requests\responses made by service clients and serving cached responses when the client is in disconnected mode. Cached requests are queued up for replay (playback) at the reintegration phase [3].

Two critical issues surrounding caching of Web Services are identified, namely Cacheability\Playback, and cache Consistency maintenance. For an effective web service cache, the authors identify the requirement that all service operations must be designable as Update or Query operations (Read\Write semantics). The lack of semantic information in the service description (WSDL) creates a challenge for caching consumers of boxed web services. Query operations are deemed cacheable if they do not alter the server state (e.g. server logs). Update operations are operations that are state altering on the server [3].

Maintaining strong cache consistency on a mobile client is deemed unachievable by the inherent property of weak connectivity. Consistency is also explored when execution of an operation invalidates a stored response of another. A proposed solution is to invalidate the old record, or apply a modification transformation for the in-cache record. The authors declare that "for preexisting Web services, understanding the correct consistency requirements is an extremely challenging issue" [3].

The effect of a web service cache on the User experience is identified as a crucial criterion when evaluating an effective web service cache. Ideally, the user should not be aware of disconnections but this is identified as a difficult goal. An additionally challenge is the ratio of consistency guarantees offered by the cache and the quality of the user experience. Altering the client to be cache-aware, and to display hints to the user regarding the active consistency guarantees may have a positive effect on the user experience [3]. The later proposition does not satisfy the property of transparent deployability outlined at the beginning of the paper.

Terry et al. discuss the effect of differing structural formatting of SOAP messages on a web service cache. This is identified as a possible problem when comparing requests for similarity. WSDL is identified as providing enough information for an intermediate proxy to fabricate a fake response to a service request. Although the lack of a specification mechanism for Default values may limit the range of possible fabricated responses [3].

Prefetching (hoarding) is identified as a mechanism to maximize cache-hit rates. An implementation would employ an algorithm for anticipating requests for prefetching. The lack of semantic information about the service and the lack of a standard mechanism for users to specify a set of requests for hoarding complicate a cache implementation supporting prefetching [3].

Finally, maintaining application Security is outlined as an important consideration. Though is complicated by lack of standards regarding authorization of access to a web service operation. An intermediary Proxy, caching for multiple clients, may open a set of privacy\security holes, this is relevant when cached responses differ by the authenticated user [3].

3.3 Caching of Objects in Distributed Object Middleware (CORBA) for Mobility: Domint

Distributed object middlewares offer remote method execution, platform interoperability, and location-transparency of objects. The first two properties offer the closest resemblance of the Web Services paradigm.

Conan et al. [2] describes an architecture offering disconnected operation of a CORBA environment for mobile clients. Domint, uses portable interceptors (PI), a CORBA mechanism for peaking into and altering of the communication between a client and an ORB (Object Request Broker). Domint offers continued operation in partial and null-connection modes with minimal or no overhead when operating in connected mode.

The transparency of utilizing CORBA's portable interceptors enables connection awareness to be shifted away from both the client and server objects and into the Domint middleware extension. Domint works by intercepting requests made to the CORBA ORB and transparently rerouting requests to a local disconnected object [2].

Several performance protective measures are employed. In order "not to punish strongly-connected clients", while strongly connected, client’s requests go directly to the remote object [2]. Also a hysteresis mechanism is proposed for handling variations in connection availability. An interface to the hysteresis mechanism maybe consumed by the client application in order to alert the user to changes in the connectivity-mode, also offering the user the ability to voluntarily disconnect. Three connectivity modes are recognized, namely: disconnected, partially connected, and connected. Transparent switching between modes is activated at the time of a client request. A set of inputs is required when deciding on an operation to execute, the inputs are: disconnection mode (voluntary or involuntary), the mode of the last request (to the same object), the operation name, and the network\object current connection mode. A matrix is developed allowing correct state transfers in various modes [2].

In the connected mode, the requests are immediately sent to the remote object. In the partially connected mode; the operation is executed both locally and remotely, depending on the call semantics (presence of in, out, in\out parameters and if a return value is expected). In disconnected mode, operations are executed locally, and are logged depending on the semantic relationships with other operations. The log is vital at the reintegration phase and reconciliation may need to occur to maintain coherency between the disconnected object (the proxy) and the remote object. However, Domint assumes that no object is accessed by more than one disconnected client [2].

Preliminary performance evaluations, performed on a Windows CE device, show overhead of 14% to 1% when connected, 50% to 6% when partially connected, and from 20% to 6% when disconnected. The incurred computation cost is justified by the introduction of transparent connectivity, without modification to either the server or the client implementations [2].

3.4 Delayed Execution\Call Aggregation: Reducing Overhead of .NET Remoting

In the context of Web Services, .NET Remoting is the infrastructure implicitly in-use when .NET applications publish or consume Web Services. Remoting abstracts remote objects to behave as local objects. The Remoting infrastructure offers various extensibility options, the lowest level communication channel maybe replaced or customized, the messages to be exchanged maybe modified before or after formatting, and calls to remote objects maybe intercepted immediately after a consumer issues a request and before the request is propagated downwards in the remoting stacks. The later is the mechanism commonly used in distributed object middlewares. The client accesses the remote object via a local proxy, known as the Transparent Proxy in .NET Remoting. The transparent proxy is generated at runtime by the Real Proxy (also a client-side object). The real proxy is generated when remote objects are referenced, and its binary maybe replaced or modified without modification to the object consumer.

Clegg [11] discusses the overhead introduced when employing .NET Remoting for remote execution. Clegg describes and evaluates an architecture that transparently monitors and optimizes calls to remote objects. RROpt is modeled on the DESORMI framework (Delayed-Evaluation, Self-Optimizing Remote Method Invocation by Kelly, Field, Bennett, and Yeung). The implementation is a modification to remoting-relevant code in the .NET CLI, namely the Mono CLI. Such an implementation eliminates the need for server\client modifications [11].

PROpt works by checking at runtime for candidate delayed calls, specifically by looking for methods of objects inheriting from MarshalByRefObject. When a remote call is incurred, it is stored in a delayed-list and a dummy return is pushed to the stack. If a method attempts to use the return value then the delayed method is immediately executed. A set of delayed methods is executed by formulating a plan encapsulating their data dependencies and forwarding the plan to the server. PROpt assumes that all servers are PROpt-enabled (executing on top of Mono CLI with PROpt extensions). Argument aggregation is also performed when a set of methods share an argument [11].

Another optimization employed by PROpt is Plan caching, sets of previously executed aggregated-calls (a plan) are remembered on the server, a client refers to them by ID, furthermore decreasing network traffic requirements [11].

The remoting infrastructure protects applications by containing them in “Application Domains” (light weight processes). Multiple client accessing the same remote object are not aware of each other, multiple application domains may be hosted within one process. Furthermore, multiple remote objects on a single remote server maybe accessed by a set of application domains in a process. In order to aggregate cross-object, PROpt implements its aggregation targets per server name [11].

The speedup possible by PROpt did not prove to be consistent. PROpt performed well when data dependencies between methods existed. Outbound Network traffic is significantly decreased due to call\parameter aggregation. PROpt optimizations failed to materialize when no data dependencies exist between method calls, this is the case when the network infrastructure is fast [11]. No applicability to mobile clients is considered.

4. A WEB SERVICE CACHE

4.1 General Considerations

An architecture supporting a predefined set of services and a special client implementation is an application-specific cache. An application-specific cache is optimized for the application logic, all caching decisions (such as placement, replacement, prefetching) target optimal consistency and performance of the application. Proprietary communication protocols (such as ones supporting compression, or multicast notification of cache invalidation) are expected, as the application permits [3, 8].

A General caching architecture, on the other hand, offers a cache to any Web service. Such architecture faces many challenges, most importantly is the decision of cacheability of a web service request. A Web service is treated as a black box, requiring the availability of cache-hints (metadata) supporting decisions such as cacheability, invalidation conditions, and default responses (return values) [3].

A caching proxy is necessary in a general cache, in order to support independent caching decisions (independent of the client and the server). On the other hand, an application-specific implementation maybe embedded in the web service and client implementations.

When the goal of a cache is to improve the availability of a web service, then a larger Cache-size and optimal placement and replacement strategies are a priority. The existence of stale-resources (invalid cache records) in the cache is permitted, hence to improve the service's availability. Knowledge of the MH connection-state is necessary, in order to seamlessly resume returning of cached responses when the MH enters null connectivity. Cached responses are returned only when the MH enters null connectivity, in order to achieve the best-possible cache consistency. Newer requests overwrite their older counterparts in the cache.

On the other hand, when the goal of caching is to improve the application's performance, then cached responses can be returned even when the MH is in full connectivity mode. Such an approach may result in substantial response-time improvements, especially for expensive web service methods: methods requiring a relatively large set of arguments, methods requiring an expensive or lengthy computation, and methods returning a sizable data object. Web service requests maybe aggregated to improve bandwidth utilization while returning cached responses. The downside to such performance optimizations is decreased cache consistency in relation to the real web service. A workaround to further improve the consistency of cached responses maybe achieved by periodically prefetching, or periodically submitting invalidation-queries of expensive cached responses.

4.2 Cache Location

A server-side cache offloads the server from re-computation of frequent requests. Such a cache implementation is commonly a specialized architecture. Cache-consistency is best obtained when using this approach, since invalidation reports maybe requested or broadcasted to the known caching proxy (or proxies). Another improvement is the transparency of the cache for a MH consuming the web service. A slight improvement in service availability is present due to a protection from server downtimes, as the server-side cache continues operation while the server is down. On the other hand, a MH suffering local null connectivity is also disconnected from the server-side cache.

A client-side cache offers the service's availability to the MH while in null connectivity [10]. Cache-consistency is minimally maintained in this approach. Near-time cache invalidation is harder to achieve since the MH cache is independent of the service implementation [3, 8].

An intermediate caching proxy offers transparent service caching for a MH. A caching proxy is more capable of tracking the list of caching MHs as it act as an intermediate delta between multiple MHs and multiple service providers. A shared cache is in effect, as requests from multiple MHs are cached for other MHs. An intermediate cache is best equipped, independently of the service provider, to deliver invalidation reports to the tracked list of MHs. The service remains unavailable to a MH in null connectivity, as the intermediate cache becomes disconnected.

A client-side cache assisted by a caching proxy is best equipped when the goal is protect the client from service unavailability while maintaining best-possible cache consistency. Several performance improvements are now possible because the client-side cache and the intermediate caching proxy can agree on a proprietary communication protocol supporting better request aggregation and near-time broadcasts of cache invalidation reports.

4.3 Semantic Metadata

The lack of semantic metadata f a web service methods presents a challenge for caching independently of the service implementation [3, 8]. The metadata may accompany the service as an extension to the service's WSDL document, or maybe maintained by a third party maintaining a repository of metadata records targeting a growing set of a web services. An alternative is to allow the service consumer to specify an updatable set of meta tags, assisting caching decisions when determining cacheability and invalidation conditions. Client maintained metadata are specified per web service.

A web service method should be tagged if it is cacheable, and a default return value should be specified. The former aids the cacheability decision of the cache, while the latter offers a protection against cache-missed of a MH in null connectivity. Additional tags may outline invalidation conditions based on time-values, age-thresholds. Method interdependencies will aid request-aggregation logic and can also provide further improvement when maintaining cache consistency by invalidating cached responses when state-modifying requests are executed.

4.4 Caching Policy

The request signature and argument values must be considered in hash function supporting cache placement of web service responses. Since multiple requests to the same method may differ on a single argument, while unique responses are always returned.

LRU, LFU and Size are competing replacement strategies when a decision relates to limiting the cache size, especially for a MH with space and computation constraints [7]. It is to be determined if any or a combination of the well studied replacement strategies are best suited for a web service cache supporting mobile devices.

A cached response maybe invalidated by its age or a timeout value. Furthermore, a cached resource maybe invalidated and evicted from the cache because an invalidating request was submitted.

Cache invalidation reports maybe broadcasted by the server or an intermediate proxy, or maybe piggybacked on the results of new requests. Furthermore, invalidation query maybe submitted on intervals or piggybacked on new requests. Limitations are present depending on the cache location and the number of entities supporting caching between the MH and the service provider [3, 8, 10].

Default return values are useful on a cold-start or when a cache-miss occurs while the MH is in null connectivity mode.

An alternative for recovery from a cache-miss when the original service is down while the MH is fully connected is by rerouting of requests to a service replica. This approach further protects the MH from service unavailability due to provider downtimes or peak hour unavailability.

The consistency of responses between the service replica and the original provider is an issue out of context for this research.

4.5 Prefetching\Hoarding

On a cold start, a MH may issue a set of predefined requests for caching. Alternatively, the service provider or an intermediate cache maybe store an up-to-date image of most frequently requests methods and push them to the client on a cold start.

Prefetching offers substantial improvements in response times to most frequently requested methods, at the expense of higher bandwidth utilization. The negative side effects of prefetching maybe overcame by request-aggregation and by adaptive prefetching logic with explicit awareness of network QoS [9].

4.6 Client Sessions

A MH utilizing a communication channel supporting session state can benefit from an intermediate cache retaining a MH-tailored list of frequent requests. Prefetch or hoarding requests can be initiated on the behalf of the MH.

For a frequently disconnecting MH, expensive requests made while in full connectivity maybe pushed upon reintegration. The minimum improvement is in effect when a cache-refresh cycle (or a cache-replacement function) is executed and an expensive request is not evicted because the owner MH connectivity is considered.

5. PROPOSED ARCHITECTURE

The design of the proposed architecture is modelled to support the following two scenarios and their consequences:

1. Null Connectivity: The MH enters null connectivity, on a new request the following conditions are evaluated:

   A] Cache-hit, the cached response is returned. A state-altering request is queued into the Replay queue. State-reading requests are queued into a Delayed-fetch queue, a mechanism for improving cache consistency at the reintegration phase.

   B] Cache-miss (Cold Start), a default return-value is returned. A state-altering request is queued in the Replay queue. A state-reading request is queued for delayed fetch.

2. Full Connectivity: The MH enters full connectivity, the following conditions are evaluated:

   Non-empty Cache:
   A] Cache Miss:
   New requests go directly to the Web Service, and when a response is received; a request\response tuple is inserted into the cache.

   B] Cache Hit: State-altering requests are sent directly to the service, a request\response tuple is cached. A response to a state-reading request is returned from cache (if the response is valid or if the request is long-living), the state-reading request is queued for delayed-fetch, a mechanism for improving cache consistency in the long-run. A long-living request is a request with a long cache TTL or the time (t) of the cached response is less than the invalidation time (On Time) of the method.

   Empty Cache (Cold Start):
   A] On start, if an intermediate proxy exist, request a cache Image, Done.

   B] On start, if a Prefetch-list is known, queue all items from the list into the Delayed-fetch queue, Done.

   C] On a new Request and a Cache Miss: Return the default return-value associated with the request and queue state-altering requests into the Replay queue. State-reading requests are queued for delayed-fetch.

For effective caching of the Web Service this approach uses a document encapsulating the Semantics of the Web Service. The encapsulated semantics are shareable, and are either defined by the service Consumer (at development time) or by the service Provider. This paper refers to the document encapsulating the service semantics as the Service Semantics Description Document (SSD Document), an XML document. The SSD also specifies Hints aiding various cache operations (Invalidation Conditions, Prefetch Lists, and addresses of a Replica and Intermediate Proxy).

A developer tool that integrates within the IDE of Microsoft Visual Studio.NET is provided to aid a mobile application developer to seamlessly specify an SSD. A similar tool is also provided for a Web Service developer in order to specify an SSD. The developer tool allows transparent incorporation of a cached Web Service while decoupling the programmer from the caching infrastructure.

The Service Semantics Description document specifies the following metadata regarding each service Method:

1] Cacheability: specifying if a method should be cached or not. A method is non-cacheable if a cached value will always lead to faulty application logic (e.g. a GetLastRequest method).

2] Replay: upon reconnection, this tag hints if a method should be replayed or not. To maintain application logic, a State-altering method should be tagged for 'Replay'. A method may not be tagged for replay because of one of two reason; either the method is state-reading or the method's state-altering behaviour is irrelevant upon reconnection.

3] Default Return Values: the value of this tag is a serialized-graph of a meaningful default return value for a method. This tag enables the MH to partially recover from a Cache-Miss while in full connectivity, or when a MH cold starts. It is expected that only cacheable (state-reading) methods will have default return values. State-altering methods can not have default return values as this may lead to illogical returns (e.g. a Bool CreateRecord() method, returning success or failure of record creation).

4] Invalidation Conditions: a cached response should be invalidated after a specified Age (in minutes), or after a certain time of day, or when an invalidating hint is received (or fetched), the latter is not implemented.

5] Method Interdependencies: for N methods, this is an N x N table specifying interdependencies between service methods. Effective Request-aggregation can only be implemented if method interdependencies are known, the current prototype does not implement request-aggregation. A cached response is invalidated if a state-altering request, that is also a dependency of the cached response, is submitted. A table lookup is used to invalidate a cached response.

6] Prefetch Parameters: a list of method names and proper arguments is specified to enable prefetching on a cold start or at periodic intervals.

7] Intermediate Cache Proxy: a URI specifying the network path to an intermediate Caching Proxy, the caching proxy interface appears as a replica of the original Web Service.

8] Service Replica: a URI specifying the network path of service Replica. A Replica is utilized when a service downtime is detected.

The implementation of the Cache is built as a Proxy of the Web Service, the Proxy exposes an interface identical to the real service, decoupling the service consumer and service provider from the cache. The Proxy implementation is essentially a disconnected Object providing transparent access to the real service while continuously monitoring service availability and network connectivity of the MH. This object is referred to as the Caching Web Service Object (CWSO).

The CWSO implements a Hashtable for storing tuples of request\response pairs. A connectivity monitor detects service availability by requesting the service's WSDL at predefined intervals (5minutes). The connectivity monitor has OS hooks to detect network connectivity at the MH, entering the CWSO into one of two states; Full Connectivity and Null Connectivity. A Replay FIFO queue and a Delayed-fetch FIFO queue are used for queuing state-altering requests (the former) and queuing of prefetch-requests (or delayed-fetch requests).

The SSD document is stored as an XML file accompanying the Real Proxy assembly. The CWSO provides an interface for consumers wishing to alter the service semantics or the default caching policy at runtime.

The Cache hashtable contains wrapped IMessage objects (CacheEntry). An IMessage object specifies the method's signature, argument list, call context, and method's response. The CacheEntry object includes the request time and time of the last cache-hit, along with the object's size in bytes.

The Cache Manager executes at state transitions (Full Connectivity and Null Connectivity) and appropriate action is taken. The Replay queue is processed before processing of the Delayed-fetch queue, to allow state changes to occur before processing of new state-reading operations. The processing of the Replay queue is started at a random interval between 1-5 minutes after achieving full connectivity. On a Cold-start, the replay-queue is empty and processing commences with the delayed-fetch queue instead.

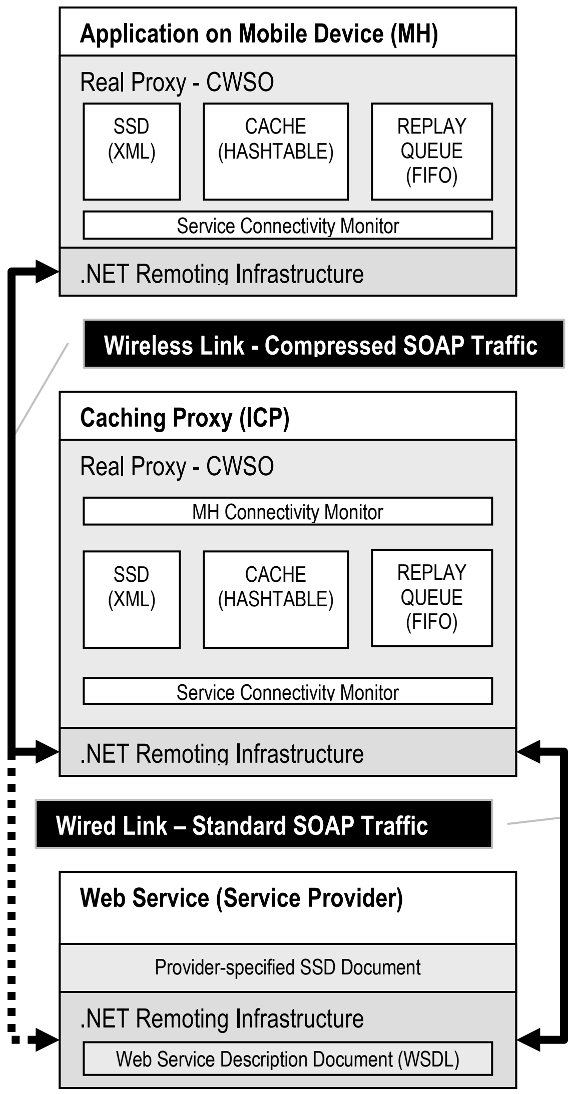
The Intermediate Caching Proxy (ICP) hosts an exact copy of the MH's CWSO, plus an additional connectivity monitor targeting MHs. Connectivity session management between the ICP CWSO and the MH's CWSO is planned as future work.

The link between the MH and the ICP is a wireless link susceptible to disconnection and low QoS (factors of weak connectivity), SOAP communication is tunnelled over a 'customizable' HTTP channel (e.g. Compressed). The link between the ICP and the real Web Service is assumed to be a wired link, offering higher bandwidth and strong connectivity, the communication is couriered by standard HTTP.

When the MH's CWSO is accessing either the real Web Service or the ICP; the logic is consistent. A replica Web Service appears to be the real Web Service to both the CWSO and the ICP. The MH or the ICP both appear as simple clients to either the Web Service or the Replica.

If a Service Connectivity Manager detects a service downtime (failure to return a WSDL document), requests are transparently routed to a Replica (if exist) or the Cache. If the connectivity manager detects null connectivity at the MH, then the ICP, the real Web Service and the Replica are all disconnected, and all requests are routed to the local Cache.

Cached responses are invalidated by Age or Time (from the SSD), the detection of an invalid CacheEntry happens when a cache-hit is suspected or when the CWSO executes the cache's SizeManager (every 20mins). CacheEntry objects maybe evicted from the cache if the cache-size exceeds a threshold (predefined as 10mb), the eviction strategy maybe LRU or SIZE.


Application on Mobile Device (MH)
Real Proxy - CWSO
SSD (XML)
CACHE (HASHTABLE)
REPLAY QUEUE (FIFO)
Service Connectivity Monitor
.NET Remoting Infrastructure
Wireless Link - Compressed SOAP Traffic
Caching Proxy (ICP)
Real Proxy - CWSO
MH Connectivity Monitor
SSD (XML)
CACHE (HASHTABLE)
REPLAY QUEUE (FIFO)
Service Connectivity Monitor
.NET Remoting Infrastructure
Wired Link – Standard SOAP Traffic
Web Service (Service Provider)
Provider-specified SSD Document
.NET Remoting Infrastructure
Web Service Description Document (WSDL)

Service Semantic Description (SSD)
CACHEABILITY TAGS: Cacheable *BOOL, Replay BOOL*
DEFAULT RETURN VALUES: *Serialized-Graphs (XML)*
INVALIDATION CONDITIONS: On *Time, After Age*
METHOD INTER-DEPENDENCIES: *N x N Table*
PREFETCH PARAMETERS: *Method & Argument List*
INTERMEDIATE CACHE PROXY: *URI*
SERVICE REPLICA: *URI*

6. PRELIMINARY EVALUATION

6.1 Experimentation Plan

Two experiments have been performed, the first experiment measures the Overhead introduced by CWSO on the MH. The second experiment captures the State Transitions when controlling factors are toggled.

The scenarios are controlled by the following factors:

- Network Connectivity: Connected and Not Connected
- Service: Available and Unavailable
- Request: State-reading R, State-altering W
- Cache Test: Hit or Miss

The service provider in the Experiment 1 is on the same host as the emulated MH. This decision was made to eliminate network latencies from the experimental results. All performance data in this experiment is collected at the local MH.

The service provider in the Experiment 2 is a remote host. The collected performance data is local to the MH.

A mobile network was not utilized in these experiments, this should not skew the result sets since this implementation does not introduce additional communication.

The link between the ICP and MH's CWSO is a standard HTTP stream, SOAP Compression is not yet implemented.

Cooperative Caching and Prefetching has not been tested.

Network, Service, Replica and ICP availability is simulated by object parameters.

The active replacement policy is LRU.

6.2 Test Suit

6.2.1 Hypothetical Web Service (HWS)

The service exposes 4 methods, R denotes a State-reading method, W denotes a State-altering (write) method:

OUT R_1() consistently returns a constant value.

OUT R_2(in) is a function of *in*.

OUT W_1(in) is state-altering returning success\failure.

VOID W_2(in) is state-altering without return.

The 'in' argument to method R_2 is an Integer, W_1 and W_2 'in' arguments are of type String, the String arguments vary randomly in size between 100bytes and 1kb.

The return value of R_1 is a String, R_2 is an Integer and W_1 is Boolean.

The associated SSD is available in [13].

The goals of this experiment is to test processing overhead (CPU) and cache-size overhead (Memory) at the MH's CWSO when 1000 requests are sequentially executed.

To calculate CPU and Memory overheads, the experiment is run twice, for each method. The first run is performed without the CWSO, the second run is with the CWSO. CWSO initialization times are accounted for.

6.2.2 I-Help Web Services

I-Help is a real-world public discussion forum system, utilized mainly by Computer Science students at our department. Actual User Traces were not collected for this experiment, instead the goal of this experiment is to verify the architecture's general applicability to existing Web Services and secondly to verify the system's State Transitions when controlling factors are toggled.

I-Help Web Services exposes two operations of interest, a Query operation and a Post operation, the associated SSD document is available in [13].

6.3 Experimental Conditions

Mobile Application: Proof of Concept Client

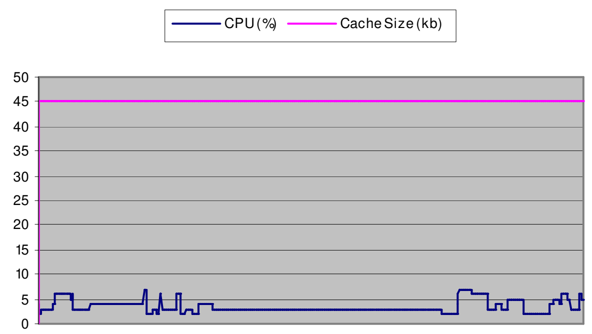
MH: Emulator Windows CE.NET 4.2.

Experiment 1 Service Provider: .NET Assembly

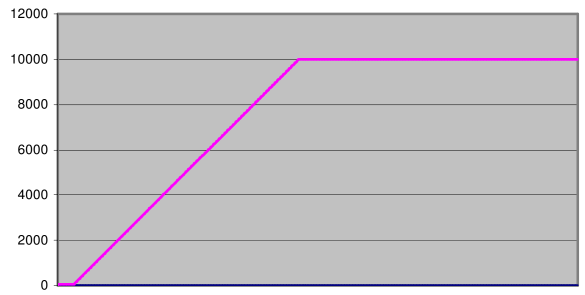
Experiment 2 Service Provider, 3rd-party implementation: Axis, Java Web Services

6.4 Preliminary Results

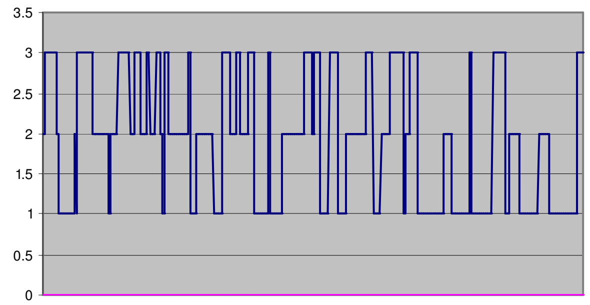
6.4.1 Experiment 1: Overhead



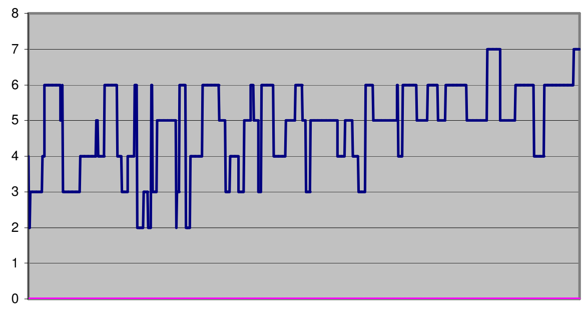
CPU and Cache-Size, Experiment 1, Method: R_1:



CPU and Cache-Size, Experiment 1, Method: R_2



CPU and Cache-Size, Experiment 1, Method: W_1



CPU and Cache-Size, Experiment 1, Method: W_2

6.4.2 *Experiment 2: State Transitions*

Network Connectivity	WS Availability	Cache Hit	Operation	Execute at Service	Cache	Default	Delay Fetch	Replay
Y	Y	Y	R		1		1	
Y	Y	Y	W	1				
Y	Y	N	R	1				
Y	N	N	W			1		1
Y	N	Y	R		1		1	
Y	N	Y	W		1			1
N	Y	N	R			1	1	
N	Y	N	W			1		1
N	Y	Y	R		1		1	
N	N	Y	W		1			1
N	N	N	R			1	1	
N	N	N	W			1		1

6.5 Summary of Results

The results show that the CPU overhead at the MH, expended in the Real Proxy object by the components: Service Connectivity Monitor, Cache-Hit Test, Cache-Size manager and the Queues, did not rise above 10%, as a preliminary result this is acceptable.

The SSD's description of W_1 as non-cacheable successfully resulted in a Cache-size of 0bytes. W_2, marked with a VOID return resulted in a similar outcome. R_1 returning a constant occupied 45kb in the Cache, a value indicative of the size of initial CacheEntry object, Hashtable initialization along with a set of data owned by the .NET Framework's memory management. On R_2 execution, cache-size grew rapidly as random 'in' arguments were sent with every new request, the cache-size capped at 10mb, the predefined maximum allowable cache-size, future requests replaced in-cache entries by the LRU strategy.

The CWSO State changes match expectations. The detected States match the architecture's logical design. This experiment utilized a real-world Web Service, demonstrating general applicability of the architecture.

7. FUTURE WORK

Embed the Service Semantics Description within the service's WSDL, this maybe done by utilizing WSDL Extensibility via attributes and element extensions. Merging the syntactic description (WSDL) with the semantic description (SSD) is very valuable, revoking the need for a separate SSD document and enabling smoother integration and richer discovery of Web Services.

8. CONCLUSION

This paper presented a generally applicable, connectivity-aware, and a transparent approach to caching of Web Services. A set of semantic tags have been identified as a prerequisite for effective web service caching. The Service Semantic Description document was developed, along with a tool enabling SSD-specification from within a widely used developer IDE. Preliminary evaluations of the architecture demonstrated general applicability, transparent operation and low resource overhead.

9. REFERENCES

[1] D. Barbar, T. Imielinski. Sleepers and Workaholics: Caching Strategies in Mobile Environments (1994).

[2] Denis Conan, Sophie Chabridon, Olivier Villin, and Guy Bernard. Domint: A Platform for Weak Connectivity and Disconnected CORBA Objects on Hand-Held Devices (May 2003).

[3] Douglas B. Terry and Venugopalan Ramasubramanian. Caching XML Web Services (May 2003).

[4] Jen-Yao Chung, Kwei-Jay Lin, Richard G. Mathieu. Web Services Computing: Advancing Software Interoperability (2003).

[5] Jia Wang. A Survey of Web Caching Schemes for the Internet (1999).

[6] M. Baentsch, L. Baum, G. Molter, S. Rothkugel, and P. Sturm. Enhancing the web infrastructure - from caching to replication (April 1997).

[7] M. Tian, T. Voigt, T. Naumowicz, H. Ritter, J. Schiller. Performance Considerations for Mobile Web Services (2003).

[8] Matt Powell. XML Web Service Caching Strategies (April 2002).

[9] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies (June 1995).

[10] Roy Friedman. Caching Web Services in Mobile Ad-Hoc Networks: Opportunities and Challenges (2002).

[11] Sam Clegg. Reducing the Network Overheads of .NET Remoting through Runtime Call Aggregation (2003).

[12] W3C Web Services Activity (http://www.w3.org/2002/ws/)

[13] Paper Appendix (http://www.cs.usask.ca/~kae501/880/)