A CACHE FRAMEWORK FOR NOMADIC CLIENTS OF WEB SERVICES

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the degree of Masters of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

KAMALELDIN ELBASHIR

ABSTRACT

This research explores the problems associated with caching of SOAP Web Service request/response pairs, and presents a domain independent framework enabling transparent caching of Web Service requests for mobile clients. The framework intercepts method calls intended for the web service and proceeds by buffering and caching of the outgoing method call and the inbound responses. This enables a mobile application to seamlessly use Web Services by masking fluctuations in network conditions.

This framework addresses two main issues, firstly how to enrich the WS standards to enable caching and secondly how to maintain consistency for state dependent Web Service request/response pairs.

ACKNOWLEDGMENTS

I would like to express my full gratitude to my supervisor Ralph Deters, for his guidance, his inspiration, and his consistent support of my journey in research. I would also like to thank my Supervisory Committee, Dr. John Cooke, Dr. Julita Vassileva, and Dr. Yang Shi for their time, their valuable feedback and support of my research.

I would also like to sincerely thank Ms. Jan Thompson for her continuous help and support, her time, and her wonderful cheerful soul that never fails to uplift.

I'd like to thank God for allowing me to overcome obstacles, for the uncountable blessings, and for enriching my life by the influence of the most wonderful people. I'd like to thank my dear father AbdelRahman Elbashir for being my best friend, and for standing by me, my friends Abubakar Benna, Ben Daniel, Brent Robertson, and Idalia Maldonado for believing in me, inspiring me, and for their extraordinary friendship.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AFS	Andrew File System
API	Application Programming Interface
ASP	Active Server Page
ASR	Application Specific Resolver
AXIS	Apache Extensible Interaction System
CDMA	Code-Division Multiple Access
CGI	Common Gateway Interface
CORBA	Common Object Request Broker Architecture
CPU	Central Processing Unit
CSD	Cache Semantics Description
DCOM	Distributed Component Object Model
DRQ	Delayed Read Queue
DWQ	Delayed Write Queue
FCFS	First Come First Served
FIFO	First In First Out
HTTP	Hypertext Transfer Protocol
ICP	Internet Cache Protocol
IDL	Interface Definition Language
IO	Input/Output
IOT	Isolation-Only Transaction
JSP	Java Server Page
MIP	Method call Interception Proxy
NU	Nomadic Unit
ORB	Object Request Broker
PDA	Portable Digital Assistant
QRPC	Queued Remote Procedure Call
RAM	Random Access Memory
RDO	Relocatable Dynamic Object
RMI	Remote Method Invocation
RPC	Remote Procedure Call

SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SP	Service Provider
TTL	Time to Live (a.k.a. Age)
UDDI	Universal Discovery and Description Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Location
WS	Web Service
WSDL	Web Service Description Language
WWW	World Wide Web
XML	Extensible Markup Language
XML-RPC	Extensible Markup Language – Remote Procedure Call
XOR	Exclusive OR

CHAPTER 1
INTRODUCTION

Background

There are a number of approaches for creating distributed object oriented systems (EMM 00). The network location of objects, in a distributed object oriented system, is hidden and the communication between objects is performed using a set of interfaces. A new approach for creating distributed object oriented systems focuses on loose coupling of distributed components by the use of existing internet standards (e.g. XML). This approach is named Service Oriented Architecture (SOA) (SCH 96, OAS 05). An implementation of SOA is a technology known as Web Services (WS) (NEW 04).

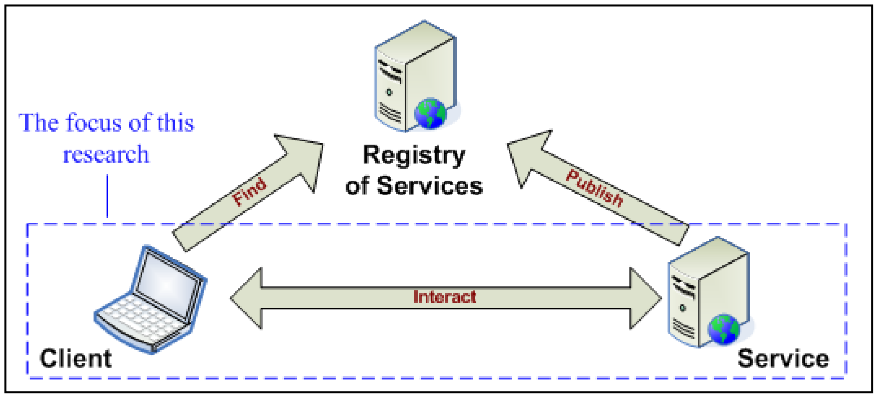


Figure 1-1. The service oriented architecture (SOA) defines three components; the registry, the service and the client components

SOA defines three components [Figure 1-1]. The components are the service (e.g. WS), the client, and the registry (e.g. UDDI[1]). SOA supports heterogeneous client and service components (SCH 96). The service component publishes self identifying information (e.g. name, organization) in the registry component. The client finds/discovers the service by performing a lookup in the registry and proceeds by interacting with the service.

An important requirement of SOA is that services are self-describing (SCH 96). The service description defines a set of operations (hereafter known as methods) and any related data types.

[1] The Universal Discovery and Description Integration, out of the scope of this research

Invocation of service methods is the primary interaction between clients and services [Figure 1-1]. After a client has performed a registry lookup and has found a service, the client processes the service description, and proceeds by invoking service methods. The pattern of method invocation/return is equivalent to the pattern of classical request/response pairs.

SOA also requires that services are self-contained and stateless. These two requirements simplify the arrangement/orchestration of independent services in order to perform a task (WEE 05). Finally, WS specify self description and interaction messages in the declarative Extensible Markup Language (XML). The use of XML simplifies the use of WS in heterogeneous environments.

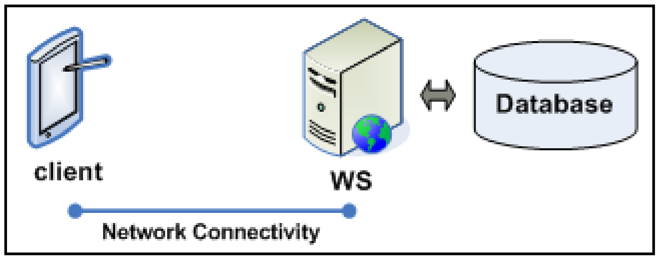


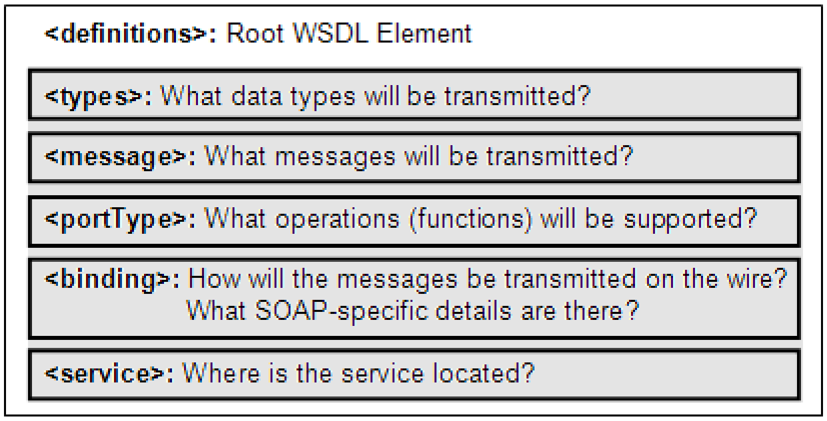
Figure 1-2. A single client and a web service, the web service (WS) uses a database

Web Services

A simple WS and a single client are used to illustrate WS and clients [Figure 1-2]. The WS implements two simple methods, namely *GetRecord* and *AddRecord*. The *GetRecord* method requires a numeric parameter which serves as a record identifier, and it returns the record's content. Further, the *AddRecord* method takes two parameters, the first is the record identifier and the second is a record's content.

The WS description is a document named the Web Service Description Language (WSDL) (W3C 05) [Figure 1-3].

The interaction between the client and WS is via a messaging protocol, named the Simple Object Access Protocol (SOAP). Figure 1-4 shows a SOAP request/response pair representing invocation/return of the *GetRecord* method. The response of a *GetRecord* request is a SOAP message named *GetRecordResponse*.



<definitions>: Root WSDL Element

<types>: What data types will be transmitted?

<message>: What messages will be transmitted?

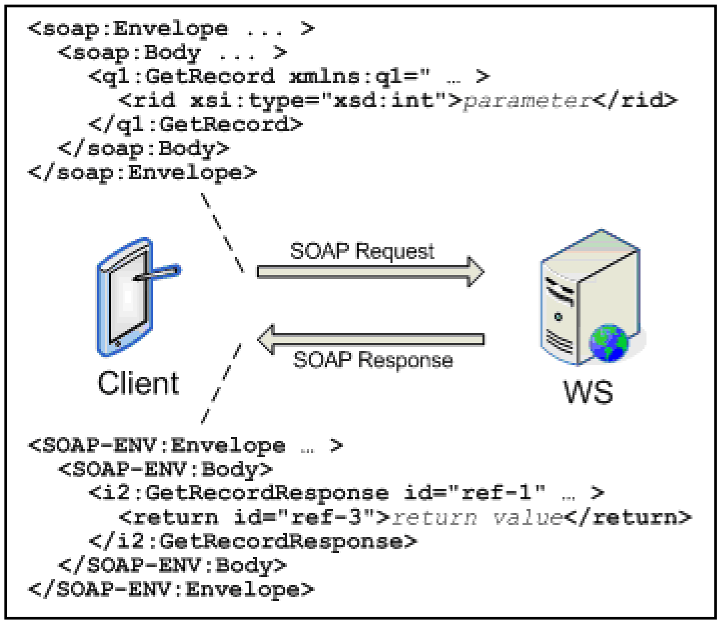
<portType>: What operations (functions) will be supported?

<binding>: How will the messages be transmitted on the wire? What SOAP-specific details are there?

<service>: Where is the service located?

Figure 1-3. The web service description language (WSDL) (CER 02)

SOAP messages are commonly[2] transported by the Hypertext Transfer Protocol (HTTP) (W3C 05). Figure 1-5 shows a SOAP message representing the outbound invocation of *GetRecord*, as transported by HTTP. SOAP's dependency on HTTP relates WS to the World Wide Web (WWW) in a relationship of common dependence.



```
<soap:Envelope ... >
  <soap:Body ... >
    <q1:GetRecord xmlns:q1=" … >
      <rid xsi:type="xsd:int">parameter</rid>
    </q1:GetRecord>
  </soap:Body>
</soap:Envelope>
```

```
<SOAP-ENV:Envelope … >
  <SOAP-ENV:Body>
    <i2:GetRecordResponse id="ref-1" … >
      <return id="ref-3">return value</return>
    </i2:GetRecordResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1-4. SOAP messages between a client and a web service

[2] Although SOAP messages are often transported over HTTP (W3C 05), the SOAP specification is transport-independent (W3C 05). SOAP over HTTP is the generally accepted and widely deployed WS scenario.

```
POST /host/service HTTP/1.1
Host: host
Content-Type: text/xml; charset=utf-8
Content-Length: length
SOAPAction: "http://host/GetRecord"

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="….. ">
  <soap:Body>
    <GetRecord xmlns="http://host/">
      <rid>int</rid>
    </GetRecord>
  </soap:Body>
</soap:Envelope>
```

Figure 1-5. A SOAP message over HTTP

The interaction between WS and client

The previous overview of the client's interaction with the WS assumed that the client has performed a lookup operation on the registry, and a service has been found. However, the overview did not provide complete detail of the interaction. The interaction can be divided in two stages, stage 1 and stage 2 [Figure 1-6]. The goal of stage 1 is to prepare the client for method invocations, while stage 2 consists of method invocations and method returns. Figure 1-6 shows a total of seven interactions, three of which (A-C) belong to stage 1 and the other four (D-G) belong to stage 2.

Stage 1 begins when the client requests the WSDL [Figure 1-6A]. After the WSDL document is received [Figure 1-6B], then a local proxy object is generated [Figure 1-6C]. The proxy object contains a set of methods which are identical to the methods of the WS. The proxy object is responsible for generating and parsing SOAP messages.

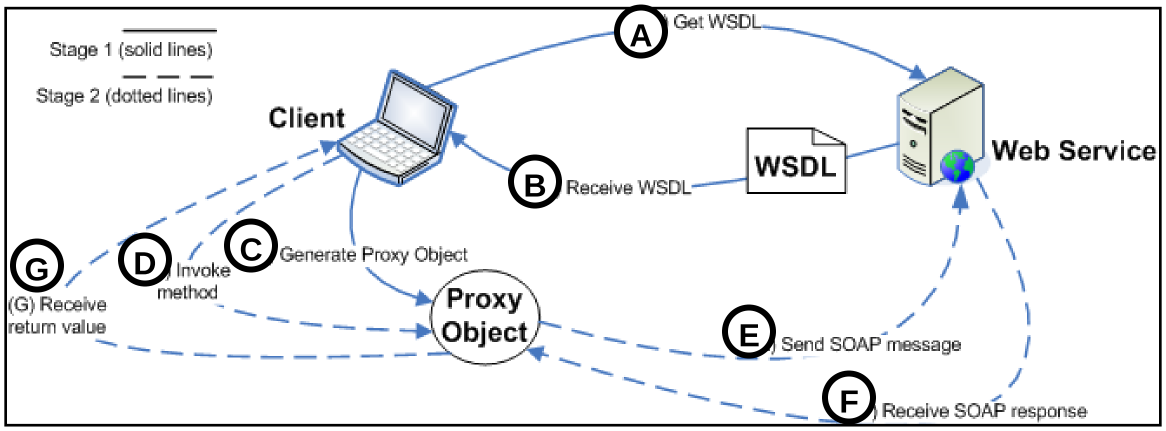


Figure 1-6. Detailed interaction between a client and WS

At the beginning of stage 2, the client invokes a method on the proxy object [Figure 1-6D]. A SOAP request [Figure 1-5] detailing the method name and parameters is generated by the proxy object and is sent to the WS [Figure 1-6E]. The WS, in turn, processes the request and returns a SOAP response to the proxy object [Figure 1-6F]. The proxy object decodes the SOAP response and creates a value which is returned to the client [Figure 1-6G].

Nomadic WS clients

Kleinrock (KLE 96) defines nomadicity as "*the system support needed to provide a rich set of capabilities and services to the nomad as he moves from place to place in a transparent and convenient form"*. The study of nomadicity focuses on the device's and application's support for mobility and disconnected operation (SAT 96, KLE 96, LAP 96). In this work, the term 'nomadic' is used interchangeably with the term 'mobile'. The nomadic WS client is defined as an application, situated on a Nomadic Unit (NU), and interacting with WS [Figure 1-7].

A nomadic WS client benefits from the loose coupling of services in SOA. The focus on nomadic WS clients is motivated by the need for mobility, coupled with the availability of increasingly faster mobile networks (e.g. IEEE 802.11 (IEE 05) and CDMA 1xRTT (CDM 05)).

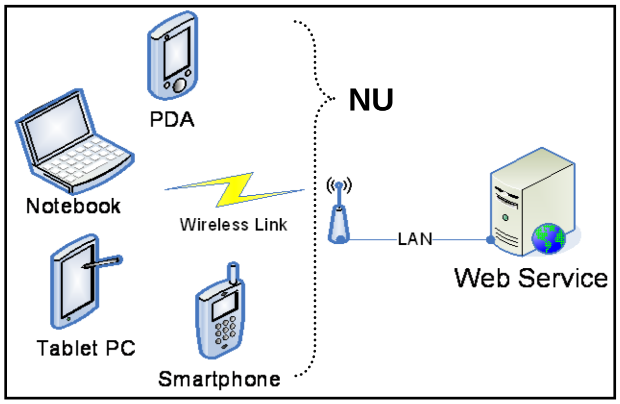


Figure 1-7. The context environment consists of mobile WS clients

The problem of intermittent connectivity

Wireless connectivity is classified as weak, since wirelessly connected devices are subject to planned and unplanned disconnections. An example of a planned disconnection is the removal of the wireless network card to conserve battery life. Unplanned disconnections are more critical than planned disconnections, an example of which is loss of the wireless signal due to user movement. The occurrence of unplanned disconnections is followed by reconnections, which is manifested in intermittent sequence [Figure 1-8].

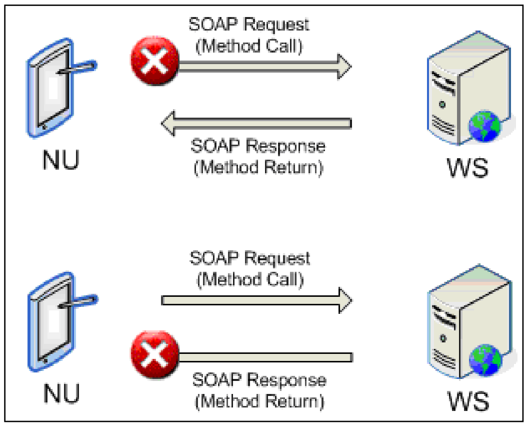


Figure 1-8. Effect of intermittent connectivity on a nomadic WS client

A WS client requires continuous availability of the connectivity resource, because SOAP over HTTP can not overcome an interruption of communication (POW 02, DEV 03, TER 03). The interruption of SOAP request/response pairs occurs in two scenarios [Figure 1-8]. The first scenario is the interruption of the outbound *GetRecord* request [Figure 1-4]. The second scenario occurs when the connectivity interruption occurs during the transmission of the inbound SOAP response (e.g. *GetRecordResponse* see [Figure 1-4]).

In either scenario, the WS client is incapable of receiving the response message, unless error handling logic is implemented. Error handling logic, using a pull approach, can resubmit the original request and then wait for a new response. Alternatively, using a push approach, the WS can resubmit the lost response when the loss of the response is detected.

However, a pull approach can lead to undesired side effects, this is the case when the invocated method is state-altering (e.g. *AddRecord*). A duplicate execution of the method *AddRecord* can lead to a corrupt or inconsistent database. The push approach, on the other hand, requires that the client is located at an unchanging (a.k.a. static) network location. This requirement can not be guaranteed in the mobile setting. The approaches for overcoming intermittent connectivity are the subject of the next section.

Overcoming intermittent connectivity

Approaches for overcoming intermittent connectivity have either relied on the replication (KIS 92) of the network component or caching (TER 03) the responses of the network component. However, replication is difficult to support when the network component is a WS. Replication requires a homogenous environment (as opposed to a heterogeneous environment). A homogenous environment assumes the equivalence of programming language, and libraries. The assumptions of homogeneity are not maintained by SOA.

An alternative solution to replication is the use of a cache, SOAP responses are cached (a.k.a. stored) for later reuse and new requests are buffered pending connectivity. Caching overcomes intermittent connectivity. This work selects a cache as the approach for overcoming intermittent connectivity of nomadic WS clients. Significant research on caches, and on overcoming intermittent connectivity motivate the selected approach.

The solution, however, is subject to the *constraints of mobility* (SAT 96, BAR 95, KLE 96, LAP 96). The complexity of the solution is constrained by the limited processing power (CPU), smaller runtime and storage memory, and a short device operation time (battery lifetime).

Caches

The concept of a cache (BAR 95, WAN 99, FRI 02) was first introduced for processor-memory communication; the concept then spread to file systems, computer networks, database systems and distributed object systems. Caches have been used to overcome disconnectivity, to decrease the latency of responses, and to increase throughput. A cache requires the identification of two cache semantics. These are cacheability and consistency maintenance.

The identity of cacheability permits a cache to store a copy of a response in the cache (e.g. *GetRecordResponse*). A response is generally not cached when the original request alters/modifies a state (e.g. *AddRecord*). In addition, a response is stored in the cache for a period of time (e.g. 1 hour), this period of time is managed by a consistency maintenance strategy. Consistency maintenance employs synchronization of state between the cache and the WS. Consistent (a.k.a. valid) cache responses are desired, they reduce the number of stale or invalid record accesses. A consistent cached value of *GetRecord* is equivalent to a value returned directly from the WS. An inconsistent cached value is known as stale/invalid. A value is stale because the record's content is modified or the record has been deleted.

A consistency maintenance strategy can delete/remove a stale response from the cache. Alternatively, the strategy can refresh the stale response by resending the original request (e.g. *GetRecord*). In this case, the new valid response overwrites the old stale response. Consistency maintenance also supports requests which alter/modify the state. A state modifying request (e.g. *AddRecord*) is known as a WRITE request. The WRITE request is logged while the network is disconnected and is submitted when connectivity is regained. The behaviour of logging a WRITE and then submitting it is known as a *Write Back*.

Problem Statement

The resource required by the nomadic WS client is connectivity to the WS. However, intermittent connectivity contradicts this requirement and motivates a solution. The proposed solution is a cache; the cache protects the nomadic WS client from the unavailability of the required resource. The problem in using a WS cache is that the cache semantics of cacheability and consistency maintenance are neither defined by the service description WSDL, nor the protocol SOAP (TER 03, DEV 03).

The challenges of the proposed solution are outlined as following:

1. Identifying the cacheability of WS methods.
2. Identifying a consistency maintenance strategy suitable for an intermittently connected, client-side WS cache.
3. The architecture is independent of WS and clients. The implementation does not require modifications of WS, WS standards (e.g. SOAP), or client applications.

Thesis Goals

The goal of this thesis is to develop and evaluate a WS Cache Model. The model defines the missing cache semantics which can support a WS cache for nomadic clients. A model consists of elements, and the complexity of elements is a component of model complexity. This thesis presupposes that the specification of more cache semantics (an increase of model complexity) has a positive relationship with efficient[3] cache operation.

1. To identify the elements of the proposed cache model: The two required cache semantics that will be used are cacheability and consistency maintenance. A simple (Optimistic Model) model tags WS methods as READs (cacheable) or WRITEs (not cacheable). Whereas a complex model represents the functional dependencies of WS methods, as a directed state graph between WS methods and resources.

2. To measure the relationship between cache efficiency and model complexity: The relationship is measured when three models, increasing in complexity (by specification of more detailed semantics) are evaluated by cache efficiency. A model is considered better if it achieves high cache efficiency at low model complexity.

3. To evaluate the client-side cost and the influence of selected factors: Model complexity influences cache efficiency, at a client-side cost. The cost is the sum of two cost sets:

- the cost of model processing: processing (CPU), and model storage (Memory)
- the cost of cache operations: processing (CPU), cache size (Memory), and network cost

Seemingly, other factors can influence cache efficiency and client cost. Two factors are selected and evaluated (one at a time); such as workload characteristics, and frequency of disconnectivity.

Thesis Overview

This thesis is organized as follows; Chapter 1 introduces the research context and described the problems and objectives of the thesis. Chapter 2 reviews important research in the areas of file caches, disconnected operation in distributed object systems, web services caches and consistency maintenance. The goal of chapter 2 is to discuss applicable approaches and how they relate to the thesis research. Chapter 3 details the implementation and design decisions of the proposed solution. Chapter 4 details the design of experiments and evaluation setup of the framework. Chapter 5 reviews the results and discusses the findings. In Chapter 6, the conclusion of this research is presented, followed by a summary of contributions, and suggestions for future work.

[3] Cache efficiency is measured by the number of cache hits and the number of consistent cache hits.

CHAPTER 2
LITERATURE REVIEW

Introduction

The architecture and standards of WS build on the standards of the web (e.g. HTTP (W3C 05)). This motivates the use of web cache literature (BAE 97, CAO 98, HUA 99, WAN 99, MAH 00) in the study and development of a WS cache. Alongside, because a WS is an implementation of SOA, the study of caching and resilience to disconnectivity in distributed object systems (CHO 00, CON 03, WAN 04) is relevant.

Caching of web documents is motivated by the need for improved availability and quicker retrieval of documents. Historically, the web consisted of documents characterized by low frequency of change; these documents are known as static. Research on static documents focused on the identification and communication of cacheability, cache policies, cache locations and hierarchies, cache protocols, and cache sharing. However, the domination of static documents was superseded by dynamic documents (e.g. CGI, ASP, JSP).

The high frequency of change of dynamic documents made it harder to identify cacheability and maintain consistency. Relevant research (CAO 98, HUA 99) focused on identifying cacheability, consistency maintenance and ensuring the availability of dependencies. The relevance of dynamic content caches to a WS cache is a result of common dependence on HTTP and the high frequency of change.

Research in distributed object systems (CHO 00, CON 03) focuses on the problem of consistency maintenance, conflict detection and conflict resolution, and disconnected operation. However, cacheability is generally assumed for methods which are state-reading (e.g. having return values).

Terminology

In this work, the terms WS client and client are interchangeably used. The client is situated on a mobile device referred to as the Nomadic Unit (NU). A cache is a memory coordinated by a cache policy, the cache content is memorized instances of data (typically responses). A cache is typically referred to by its network location. A cache residing on the client is a client-side cache, other caches are server side caches, and proxy caches. A proxy is a network component located either on an intermediary node or on the client-side. This work refers to a *live* component/resource, meaning that the component/resource is reachable.

The network traffic between the client and the server is overseen and\or controlled by the proxy. A proxy can be caching (e.g. caching proxy (CAO 98)) and a proxy can alter the network traffic (e.g. network filter). Furthermore, a proxy can be a load balancing gateway, or a buffer to an intermittently available resource (e.g. WS).

A cache is typically used to exploit temporal locality. A locally found response is quicker to retrieve than waiting for a new response from a remote location. Temporal locality is the principle stating that a recently used response will likely be needed again. A cache can also be used as a shield against intermittent availability of a network component (e.g. the WS). The cache becomes a shield when all requests are answered from the cache once the *live* network component becomes unreachable.

A cache's answer to a request is either a cache hit or a cache miss. A cache hit indicates that the response is found in the cache, and a cache miss indicates that the response is not found in the cache. A hashfunction is responsible for identifying if a new request matches an old request. The hashfunction's result identifies the location of the response entry in the hashtable, the hashfunction's argument being the origin request.

A match between requests *A* and *B* is found when the evaluation of the hashfunction on a request *A*, *hashfunction(A),* yields the same result as *hashfunction(B)*. The equality of the results means that requests *A* and *B* are equal. Consequently, the response of request *A* is suitable for the new request *B*.

Cacheability of a request is the property stating that responses can be cached without the creation of an undesired program state (FRI 02, TER 03, LAP 96). Cacheability is always true for a request that is state-reading and non state-altering. The cached response, however, remains in cache until it is determined as invalid, or until the cache size is too large and the response was not recently used. A response is invalid if it has expired by an age value (e.g. HTTP time-to-live). An invalid response must be requested again from the *live* resource.

A cache is supported by strategies for maintaining equality between local responses and *live* responses. Consistency maintenance is the term used to describe these strategies. Hence, cached responses should closely resemble the responses of the *live* resource (e.g. WS). A strategy (e.g. implicit invalidation) removes or refreshes a cached response when a condition is true (e.g. response age is more than maximum age).

Requests which read the state of a resource are called READ requests (or READs). A READ request is synonymous with the terms query, state-reading, and state-dependent. Requests which alter (a.k.a. modify) the state of a resource are called WRITE requests. A WRITE request is synonymous with the terms update, and state-altering. A client receiving a local reply to a READ has performed a read-local operation. However, a client receiving a remote response to a READ has performed a read-through operation. A delayed state-alteration is known as a write back. However, an immediate state-alteration on the *live* resource is the result of a write-through operation.

The following sections review research literature about disconnected file caches, static and dynamic web caches, distributed object caches, consistency maintenance, and WS caches. The review concludes by a summary outlining the findings.

Disconnected File Caches

The Coda file system (KIS 92, SAT 96, NOB 97) represented an important step towards disconnected file IO operations. Coda is a networked file system that evolved from the Andrew file system (AFS). Coda is motivated by the need for disconnected information access. The Coda project and related research emphasizes the need for adaptation as a result of mobility constraints (NOB 97).

Two adaptation strategies are defined, a*pplication transparent* adaptation and *application aware* adaptation (NOB 97). In the former strategy, the file system is responsible for managing information availability and control, while in the latter strategy an application can influence adaptation. The influence of the application is made possible by *application-specific resolvers* (ASR). A resolver is a file format handler, which performs conflict resolution (KIS 92, SAT 96, NOB 97).

Coda's main strategy, application transparent adaptation, has three modes of operation; hoard, emulate and reintegrate. In the hoarding mode, requests are returned from the server and are locally stored. In the emulation mode, all requests are answered from cache, and WRITE requests are logged. Finally, in the reintegration mode, Coda attempts to detect conflicts and perform resolution/reconciliation. The reintegration mode begins when the connection is regained. Conflict resolution in the reintegration phase uses Isolation-Only transactions (IOT). An IOT detects read-write conflict based on serializability constraints. However, the detection and resolution of write-write conflicts is left to ASRs (NOB 97, KIS 92, SAT 96).

Reintegration occurs in two phases, r*apid cache validation* and t*rickle reintegration*. In the rapid cache validation phase, the NU sends a list of stamps, representing cached content and the server is responsible for acknowledging or revoking the stamps. However, in the trickle reintegration phase, the NU propagates updates to the server in the background of new requests. Reintegration is further optimized by a *logical divider* mechanism, where records are either removed (not propagated) or stored in the disconnected log depending on ongoing reintegration results (NOB 97, KIS 92, SAT 96).

Coda implements user assisted cache miss handling by the use of a user model. The model approximates user patience and decides if a cache miss is reported to the user or is answered from a remote location. A remote answer has a weight proportional to the detected network bandwidth.

The ROVER toolkit (JOS 97) is a framework for the disconnected operation of objects. ROVER assumes optimistic concurrency control, and performs application transparent/aware adaptations comparable to those of Coda (KIS 92, SAT 96, NOB 97). However, the cache content of ROVER is objects, instead of files (JOS 97). The objects are named *relocatable dynamic objects* (RDO).

An RDO is replicated on the client-side and all operations are locally performed. A log of *mutated* (a.k.a. state-altered) objects is maintained by the client and is sent to the server when new RDOs are imported. A client-side scheduler is responsible for incremental push transfer of the logs. The server, in turn, is responsible for conflict detection, resolution, and the distribution of messages. The messages notify clients of all resolved conflicts.

ROVER implements queuing of remote procedure calls (referred to as QRPC). A QRPC performs lazy/asynchronous import of RDOs. Additionally, a QRPC implements a *split-phase* operation, where an interrupted response triggers server reattempts. The split-phase operation removes the need for the resubmission of unanswered requests (JOS 97).

File caches were originally developed for fully disconnected operation. The hoarding mechanism in Coda, and the replication of an RDO in ROVER are mechanisms which maintain disconnected availability of files. However, cache misses in such environments are always fatal (FRO 98). On write back policies, Froese et al. (FRO 98) observes that an aggressive write back policy conflicts with the priority of READs. In a simulation study of file write back policies, Froese et al. (FRO 98) find that a delayed write back policy does achieve good levels of cache consistency without sacrificing READ throughput.

Web Caches

In the area of web caches, and the context of static web content, Wang (WAN 99) recognizes the use of http proxy servers and client-side caches as the most common caching approaches. Proxy servers and client-side caches are nodes, which reside on the connectivity tunnel between HTTP clients and servers. Either approach operates transparently as a result of cache support in HTTP. Proxies and client-side caches are found useful because of reduced bandwidth consumption, lesser server load, and a shorter latency of responses. Additionally, client-side caches are noted for the benefit of disconnected availability of content. Wang (WAN 99) and Mahanti et al. (MAH 00) attribute these benefits to cache locality. Locally answered requests are returned faster, without the requirement of server processing. Hence, bandwidth consumption is reduced because of the elimination of a network roundtrip. Additionally, local availability of responses protect the client from server/network unavailability as requests are locally answered.

Web caches are possible due to the support for cacheability identification and consistency maintenance in the HTTP specification (W3C 05). An HTTP response specifies if the body of the response (a.k.a. the content) is cacheable. The response headers also specify an expiry time, after which the content is invalid/old (a.k.a. stale). An HTTP client can perform explicit invalidation by sending a special validation request to the server. The request is marked by a condition which states that a server response is needed only if the content has changed since a given time. Alternatively the HTTP client can send a request which queries the last time of content change, for a given request. In turn, the client compares the time of change to that of a locally cached response. The result of this comparison is that a local response is either stale or valid, and processing proceeds accordingly.

Of note, is cache sharing between HTTP proxies. Cache sharing is the responsibility of related protocols (WAN 99) (e.g. the Internet Cache Protocol ICP (CAO 02)). ICP supports interactions (a.k.a. negotiation) between caches, and is out of the scope of this work.

Consistency maintenance

Wang (WAN 99) states that the selection of a consistency model depends on the frequency of content change, and on the client's tolerance for stale content accesses. Wang (WAN 99) notes that stale content is commonly served to clients while the web server is unreachable. HTTP’s support for the maintenance of cache consistency (a.k.a. coherency) is implemented by two commonly used models. The first is the strong consistency model, and the second is the weak consistency model. Strong cache consistency is maintained by client validation (a pull mechanism), or server invalidation (a push mechanism). Weak consistency, however, is maintained when the cached content is invalidated by a value (e.g. time-to-live value).

In a pull mechanism, the result is higher cache hit latency. The latency of the cache hit is higher because the client must validate a cached item by sending a special request to the server. The push mechanism, however, requires server cooperation and a static/constant location for the client. The HTTP server is responsible for broadcasting invalidation messages (a.k.a. reports); the messages notify clients of content changes.

Wang (WAN 99) reviews a proposed mechanism known as piggyback invalidation. Piggyback invalidation is a hybrid pull/push mechanism, where the server appends an invalidation report/list to responses of new requests. The piggyback mechanism removes the push requirement of a static/constant client location. Moreover, the piggyback model is noted for resulting in a level of cache consistency that is higher than weak consistency, and lower than strong consistency.

Dynamic Content Caches

Dynamic content (e.g. CGI, JSP, ASP) are characterized by high frequency of change. Moreover, a dynamic resource is determined by both a URI and the parameter values of the URI. As a result; cacheability identification and consistency maintenance is more challenging (CAO 98, HUA 99, WAN 99).

Dynamic content caches generally follow one of two approaches (WAN 99); the active caching approach (CAO 98) and the server accelerator approach (HUA 99). Either approach imposes the use of a programming language (e.g. java) for implementation of dynamic content. The requirement of language compatibility limits the use of a cache to homogenous environments (CAO 98, HUA 99, WAN 99).

Server accelerators require the alteration of dynamic content to use special cache API. The cache API is defined by the *accelerator* entity, on the server side. The applicability of the server accelerator approach to a client-side WS cache, as such, is not practical. Hence, the following section ignores the server accelerator approach, and reviews the active caching approach.

The active caching approach

Active caches require the implementation of computation logic, known as applets. The applets are requested by a caching proxy, along with the applet dependencies (e.g. data file). The proxy, in turn, receives a request and invokes the cache applet with a defined set of parameters (e.g. request headers). The response of the applet is a new page, or a *hint* to use a cached page, or to pass the request to the server.

The cache content in an active cache is executable logic, instead of web documents. However, executable logic is commonly dependent on a data resource (e.g. database), this dependency is generally unsupported by active caches. However, Huang et al. (HUA 99) proposed an architecture where an applet implementation uses a special file IO library. The library performs automatic identification of applet dependencies (generally referred to as server data). The library intercepts file commands (e.g. open, read) and dynamically retrieves the required files.

Huang et al. (HUA 99) argue that trust between servers and caches is a required prerequisite for availability of applet dependencies. Trust is required because server data can contain sensitive organizational or user data. Trust is established after mutual authentication between servers and caches. Only a trusted cache is permitted to receive server data.

Consistency maintenance

In the approach of Cao et al. (CAO 98), a cache applet records a log of client requests, and the log is periodically pushed to the server. Additionally, resource management policies are defined. A policy categorizes the output of a cache applet as negotiated or non-negotiated. The difference between the two categories is that a negotiated document is a trusted document and the data it requires is retrieved for local storage. Trust is established if the server shares a set of HTTP protocol extensions.

The approach of Huang et al. (HUA 99) assumes cache applets are read only. Read only applets do not modify their data. Hence, implicit invalidation of applets and applet dependencies is considered sufficient.

Distributed Object Caches

Conan et al. (CON 03) describe Domint as a system enabling disconnected operation of CORBA objects for mobile clients. Domint's architecture is based on *portable interceptors* (PI), a CORBA mechanism for monitoring and altering of communication between a client stub and an Object Request Broker (ORB). Requests are sent directly to the remote object while connectivity persists. Domint's recognizes two levels of connectivity; partial and full. Domint causes minimal or no overhead in either level of connectivity (CON 03). Method calls which are executed during partial connectivity are executed both locally and remotely.

Conan et al. (CON 03) state that disconnected operation is dependent on method call semantics. The semantics of a method call are the presence of in (e.g. data), out (e.g. pointer), in/out parameters (e.g. pointer w\value) and if a return value is defined. All locally executed operations in disconnected mode are logged. The logging of method calls is dependent on the semantic relationships across method calls. Conan et al. (CON 03) identify the *reintegration* phase as the time period following reestablishment of connectivity. Logged operations are used for reconciliation (a.k.a. consistency maintenance) with the ORB during the *reintegration* phase.

Domint (CON 03) assumes that no object is accessed by more than one disconnected client at a time. The assumption of optimistic concurrency removes the complexity of distributed consistency maintenance.

CASCADE (CHO 00) is a distributed object caching architecture which supports disconnected operation of CORBA objects. CASCADE is similar to Domint (CON 03) in its use of portable interceptors (PI). A PI intercepts and alters the communication between objects. Moreover, CASCADE caches the distributed object (code) and its dependencies (data). Cached objects are organized into hierarchies based on FCFS arrival of requests. Strong consistency is maintained by imposing an order on updates, reflected in the hierarchical tree structure of the clients. The tree structure is distributed, and is also replicated at each node. The distribution and replication of the tree structure protects CASCADE from node/tree-branch failure.

In CASCADE (CHO 00), two classes of configurable policies exist. The first class is policies per request and the second is policies per object. The policies implement consistency guarantees, persistent behaviour along with more cache semantics. The per request policies are dependent on call semantics. However, the per object policies are dependent on the hierarchy of clients.

Distributed reconciliation is also implemented, and it is dependent on the selected policy class. A policy influences update propagation and the consistency guarantees. CASCADE supports a set of READ/WRITE policies; *read your writes* (a.k.a. blind reads), *monotonic reads* and *monotonic writes* (a.k.a. no conflict), *writes follow reads* (a.k.a. read preference), and *total ordering* (a.k.a. global sequencing of updates). Additionally, CASCADE guarantees eventual update propagation (CHO 00). The complexity and range of policies supported by CASCADE enable the guarantee of strong cache consistency (CHO 00).

Cache Consistency Maintenance

Wang et al. (WAN 04) developed SACCS, a hybrid consistency maintenance scheme for nomadic environments. Existing approaches are categorized as *stateful* and *stateless*. Stateful approaches (e.g. explicit invalidation) require server cooperation, while stateless approaches (e.g. implicit invalidation) do not. SACCS is a hybrid stateful and stateless algorithm. The algorithm uses a set of flags located at the server and the NU. Invalidated cache content is replaced by a placeholder ID. Upon re-connectivity, valid content is flagged as uncertain and refreshed accordingly. SACCS removes the requirement of periodic broadcasts of invalidation messages/reports (WAN 04).

In a study of consistency maintenance mechanisms for nomadic environments, Lindemann et al. (LIN 03) propose a new mechanism named lazy invalidation. Lazy invalidation does not require direct connectivity/communication between the NU and the server. The approach works by exploiting the concept of epidemic distribution of information. Epidemic distribution is derived from biological environments, where information flows between mobile nodes when they are in close contact.

Invalidation messages are epidemically distributed between nomadic units. However, a result of mobility is that lazy invalidation can not guarantee that all nodes receive the invalidation messages. Lindemann et al. (LIN 03) propose the coupling of lazy invalidation with implicit invalidation for overcoming this limitation. The proposal guarantees the invalidation of cached content, either by lazy invalidation or implicit invalidation. In a simulation study, the coupling of implicit and lazy invalidations resulted in a cache consistency efficiency of 95% (LIN 03).

SOAP Caches (a.k.a. WS Caches)

Terry et al. (TER 03) identified the need for WS caches for mobile WS clients. Transparent deployment and general applicability are identified as two desired properties of a WS cache. Transparent deployment is defined as non interference with the WS or client implementation. General applicability is defined to differentiate a cache from a service-specific and application-specific error handling logic.

Terry et al. (TER 03) recognize two critical issues in WS caches, namely the identification of cacheability and consistency maintenance. The lack of necessary semantic information in the service description (WSDL) is identified as the primary challenge for caching of *boxed* WS. The authors state that a requirement for a WS cache is that all operations are designated either as *update* (a.k.a. WRITE) or *query* (a.k.a. READ). The association of write/read semantics to WS operations identifies cacheability. Query operations are cacheable because they do not alter the service state, while Update operations are deemed *selectively cacheable* with the requirement of *replay*.
Terry et al. (TER 03) state that the requirement of strong consistency conflicts with the existing WS standards. The existing standards do not support Stateful consistency maintenance approaches (e.g. *pull, push,* or *piggy-back*). The alternative stateless approaches (e.g. implicit invalidation) are suggested as a compromise. Terry et al. (TER 03) recognize that execution of an *update* operation invalidates a stored response of a *query*, which motivates additional consistency maintenance if the state interdependencies are known. Terry et al. (TER 03) propose the use of a *modification transform* to cached records. The transform modifies cached *query* requests to reflect the state changes of an *update* invocation. The transform requires knowledge of the state interdependencies of WS method, and additionally requires executable logic approximating the operation of service methods.

Terry et al. (TER 03) declare that "*for pre-existing web services, understanding the correct consistency requirements is an extremely challenging issue*". The proof of concept implementation demonstrated the feasibility of a WS cache operating in disconnected mode. Terry et al. (TER 03) also identified that SOAP caches can encounter difficulties when attempting to match SOAP requests to previously seen requests. The difficulty leads to a false cache miss and is known as a cache busting problem. The problem is a direct result of SOAP's XML grammar; it is manifested when two SOAP headers mismatch, across two identical requests.

Takase et al. (TAK 02, TAK 04) offer a solution to the cache busting problem in SOAP caches. Canonicalization is proposed as an approach to guarantee request matches based on an XML canonicalization template. The canonical form of a document describes the permissible changes in document structure, while the document meaning is preserved. If two canonical forms match, then the documents are equivalent regardless of the difference in their physical structures. A canonicalization template is service specific and caching is demonstrated using only a modified reverse web proxy. Takase et al. (TAK 02, TAK 04) generalize all SOAP requests as read only and no special attention is given to the feasibility of the XML intensive approach on constrained mobile devices.

In a study of client-side WS caches, Devaram et al. (DEV 03) considered RPC-style SOAP messages. An RPC client is profiled and the generation of SOAP requests is divided into *stages*. The client is found to spend a considerable fraction of the request latency in the XML encoding stage. The XML encoding stage consumes 40% of the request latency. The proposed solution is SOAP compression. Devaram et al. (DEV 03) found that SOAP compression is more costly (in latency) than the XML encoding stage. Additionally, Devaram et al. (DEV 03) observed that the client-side approach leads to improved response times, as the client does not incur the cost of encoding previously encoded SOAP requests. Another observation is that a large fraction of an encoded document remains constant across requests. The authors suggest the use of these observations when deciding on the cache location and the caching unit. Devaram et al.'s (DEV 03) primary focus, however, is on the reduction of SOAP latencies.

Tian et al. (TIA 03) also observe the high cost of XML processing. The authors approach the problem by dynamic compression of SOAP messages. The server chooses the compression level depending on the client's choices and depending on server load. The method is classified as a quality of service approach. Tian et al. (TIA 03) promote the use of quality of service (QoS) approaches for mobile WS clients.

Alwagait et al. (ALW 04) describe DeW as a framework for dependable operation of web services. DeW supports location independence of the WS in respect to clients. In DeW, a WS is replicated across network nodes with the goal of providing fault tolerance and load balancing. An extended WSDL is developed along with a specialized client proxy. The client proxy does not interfere in the communication between the client and the WS. However, when a network fault occurs (e.g. WS is unavailable), the proxy is responsible for communicating with the DeW infrastructure and redirecting client requests to a WS replica. The DeW framework requires strong connectivity of DeW components; as a result it is unsuitable for intermittently connected clients. Furthermore, the DeW framework requires a SP to implement special service logic in support of DeW operation (ALW 04).

Summary

A basis for the development of a WS cache is the operation modes of the Coda file system; hoard, emulate, and then reintegrate (KIS 92). Moreover, Coda's adaptation strategies have been used by other frameworks (e.g. ROVER (JOS 97)). Application aware/transparent adaptations are guidelines for a scalable, and a heterogeneity-supporting WS cache.

The studies of web caching identify proxy caches and client caches as proven mechanisms for protecting clients from server unavailability. Existing research has identified that an application's tolerance for stale resource accesses decides the choice of a consistency model (e.g. strong vs. weak). The approaches of caching dynamic web content require the replication of executable logic. The replication of executable logic, however, conflicts with the assumptions of heterogeneous environments (e.g. the WS environment). Furthermore, replication of executable logic on mobile devices is an additional burden, because of the limited processing and storage resources.

The reviewed distributed systems (Domint (CON 03) and CASCADE (CHO 00)) show that a remote object's operation in disconnected mode is possible. The consistency maintenance mechanisms utilize a log enabling reconciliation at the reintegration phase. In these systems, the relationships between operations highly influence the cache behaviour. The relationships are the availability/sharing of in/out parameters and existence of return values. Additionally, CASCADE (CHO 00) implements cache policies which are assigned per method or per object. The use of a policy class (e.g. per method) enables the choice of higher or lower consistency guarantees. The choice of consistency guarantees is dependent on the application requirements.

Implicit and explicit invalidation of cache content are well studied. The use of implicit invalidation is shown to be the approach of choice for maintaining weak cache consistency on mobile clients. However, a hybrid algorithm called lazy invalidation is shown to offer considerable improvements to mobile cache consistency without the requirement of server cooperation.

The HTTP (W3C 05) protocol does not support identification of cacheability for HTTP POST requests. Moreover, support for consistency maintenance in HTTP uses a pull based approach. On each request, a client is responsible for testing a *last-modified* header. This mechanism, however, is only for GET requests, and it requires strongly connected clients. Moreover, state-altering requests require the minimum of delayed execution. However, the HTTP protocol does not permit/support write back of POST requests.

With regards to WS caches, the community is beginning to focus on the challenges. The challenges are in the identification of cacheability, and in the choice of consistency maintenance strategies. The lack of semantic information (a.k.a. metadata) describing the service methods is a critical limitation of the WSDL. The semantic information describes WS methods as READ or WRITE.

On HTTP, the identification of cacheability of WS methods is not possible. The HTTP specification supports cacheability headers by resource. An HTTP resource is identified by a URL. However, SOAP requests target a single resource URL, regardless of the intended WS method.

For disconnected consistency maintenance, a transform which modifies cache records is proposed (TER 03). The transform (a.k.a. modification transform) modifies a cached READ to reflect a change of state which resulted from a disconnected WRITE. The modification transform, inherently, requires knowledge of the functional dependencies between WS methods. Additionally, the transform is an executable logic, which approximates the operation of a WS method. However, the requirement of executable logic limits the applicability of the approach to homogeneous environments.

On SOAP caches, the identity of what is cached (a.k.a. cache unit) is a prerequisite for cache implementation. A cache can be implemented as a hashtable where request/response pairs are stored using a hashfunction of the SOAP request. SOAP complicates caching because it is specified as in verbose XML grammar. It is common that string comparison of two identical SOAP requests does not succeed. A hashfunction which directly compares the string representations of SOAP messages can fail. The failure occurs when two hash values, of two identical SOAP requests, are unequal. The inequality is a result of SOAP’s verbose XML grammar. A an example is when a SOAP header contains data that changes across requests (e.g. request time, or request count). As a result, two identical requests have two different string representations, and two different hash values. This is known as a cache busting problem. Cache busting results in false cache misses which decrease cache utility and efficiency.

Eliminating cache busting is important for an efficient cache implementation. A generalized XML grammar (e.g. canonicalization template) is proposed (TAK 02, TAK 04) as a method which eliminates cache busting. However, a mechanism similar to CORBA’s portable interceptors (PI) (CHO 00, CON 03) may also prove useful in this objective. A PI intercepts method invocations instead of protocol messages. Similar approaches to a CORBA PI have been used for dynamic content caches (e.g. Huang et al. (HUA 99)) and file system caches (e.g. Coda (KIS 92, NOB 97)).

CHAPTER 3
THE WS CACHE FRAMEWORK

A client-side cache is the execution environment of the WS cache model. The cache implementation, named CRISP, is a proof of concept which enables experimentation. The name CRISP is not an acronym. The system architecture which motivated the design of CRISP is outlined in Figure 3-1. The client-side WS stack (2) performs the operations of the WS proxy as originally shown in Figure 1-6. The client application (1) is unaware of the network connectivity or the communication between the client-side stack and WS (6).

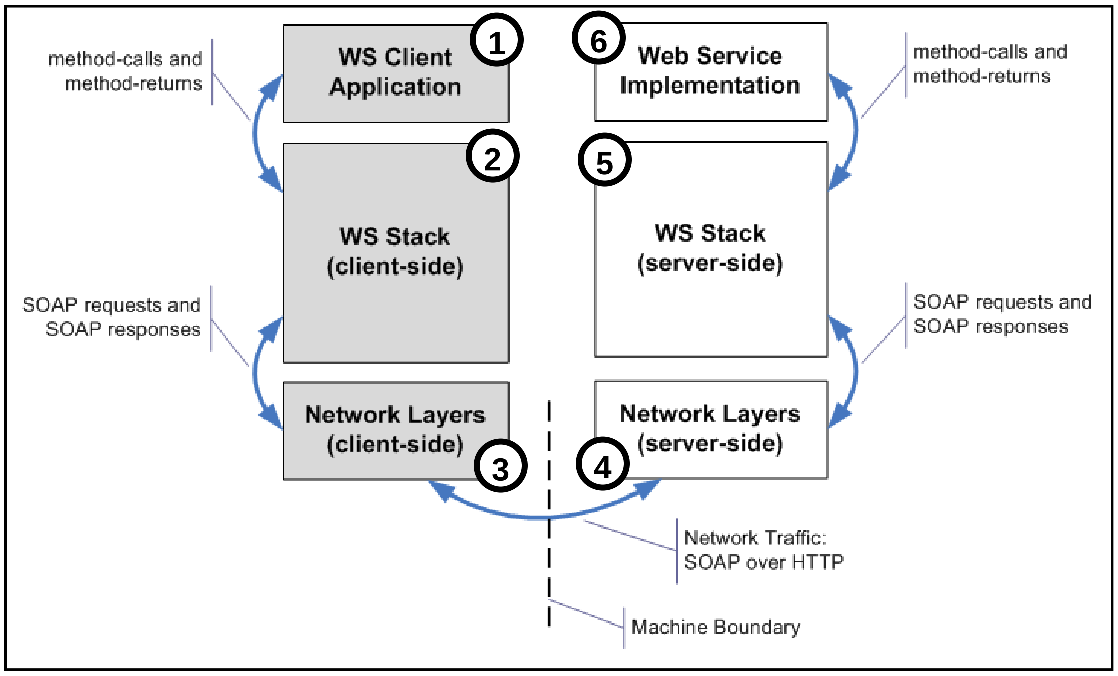

method-calls and method-returns
WS Client Application
1
6
Web Service Implementation
method-calls and method-returns
2
5
WS Stack (client-side)
WS Stack (server-side)
SOAP requests and SOAP responses
SOAP requests and SOAP responses
Network Layers (client-side)
Network Layers (server-side)
3
4
Network Traffic: SOAP over HTTP
Machine Boundary

Figure 3-1. The system architecture, the client WS stack (2) is selected as the location of CRISP

This chapter is structured as follows; section 3.1 describes the design of the client-side cache and details the roles and interactions of components. In section 3.2, the design of the WS cache model is described and an example is used to show a cache model specification. The example used builds on the simple service described in chapter 1. In all diagrams; squared brackets are used to indicate a condition (e.g. state) for an interaction. The literal M denotes a method signature and parameters values. The literal RV denotes the return value, and DV denotes a default value. The symbol *hf* is the hashfunction used as *hf(M)*.

The Client-side Cache

CRISP is a WS cache located inside the client-side WS stack [Figure 3-1 (2)]. CRISP requires the specification of the WS cache model. The model is specified in an XML document, named the *cache semantics description* (CSD).

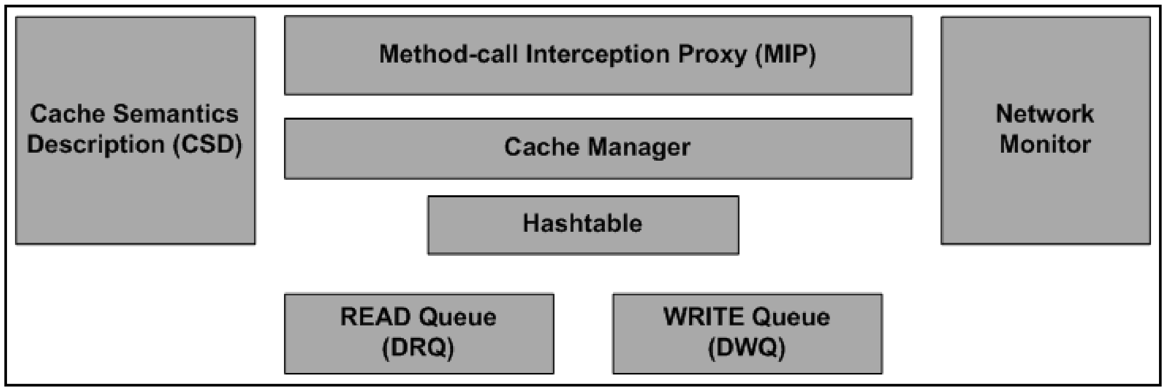


Figure 3-2. The architecture of CRISP

A simple architecture diagram is in the Figure 3-2 and a detailed version is at the end of this section [Figure 3-8]. In CRISP, cache operation is initiated when a method invocation/return is intercepted (e.g. *GetRecord/GetRecordResponse*). CRISP does not intercept or modify SOAP messages, and as a result, it does not interfere with the WS standards. Additionally, the location of CRISP within the WS stack allows transparent operation. The operation of the cache is transparent to both the client and the WS. Therefore, the WS client overcomes intermittent connectivity by transparent adaptation (NOB 97).

CRISP operates in one of three states, namely *harvest* (while connected), *emulate* (while disconnected) and *reintegrate* (once a connection is re-established).

Design decisions

The chosen location for CRISP is the client-side WS stack. The relationship between the WS stack and the client application is the exchange of method invocations/returns. The WS stack precedes the SOAP processing stage (DEV 03, TIA 03). Hence, CRISP intercepts method invocations/returns, and no SOAP messages are intercepted or modified. Therefore, the chosen location protects the hashtable from cache busting (TER 03, TAK 02, TAK 04). In addition, the chosen location means that cache operation is performed at a step preliminary to the SOAP encoding stage. As a result, the client application benefits from the elimination of SOAP processing at each request.

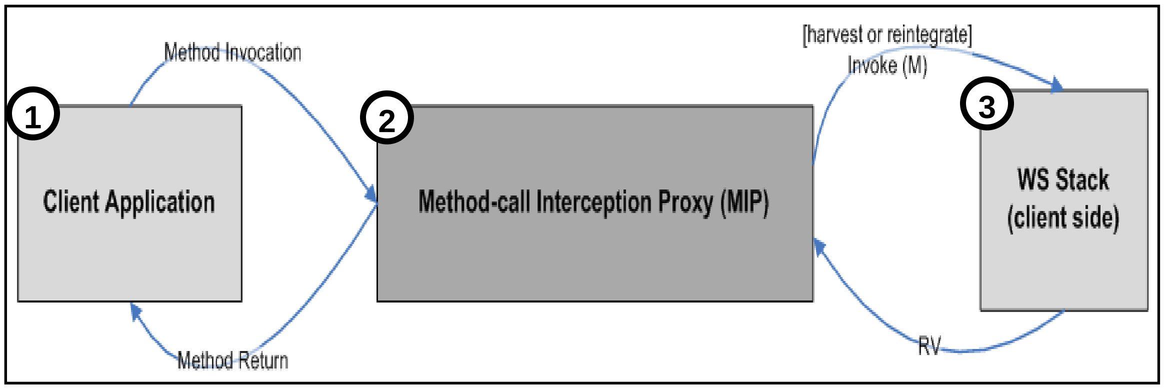


Figure 3-3. The method-call interception proxy MIP (2) is located between the client application (1) and the client-side WS stack (3)

The Method-call Interception Proxy (MIP)

The client application [Figure 3-3 (1)] invokes method calls on the WS stack. Upon each invocation, the MIP [Figure 3-3 (2)] intercepts the method invocation (e.g. *GetRecord*) and initiates the cache operation. If a request can not be answered locally, then the MIP sends the request to the lower levels of the WS stack [Figure 3-3 (3)]. The WS stack, in turn, communicates via SOAP messages over HTTP with the remote WS [Figure 3-4].

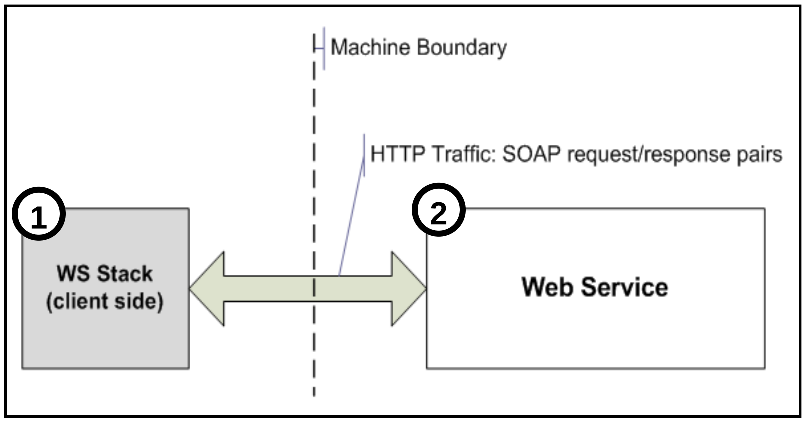


Figure 3-4. The client-side WS stack (1) and the WS (2) communicate via SOAP request/response pairs, the communication is transported by HTTP

The Network Monitor

The network monitor component [Figure 3-5 (1)] observes the connectivity state [Figure 3-5 (2)] and initiates resilience to connectivity fluctuations. Connectivity is seen as either online (ON) or offline (OFF). The state of CRISP is altered depending on the relationship between the current connectivity state and the previous connectivity state [Table 3-1].

Table 3-1. The state of CRISP is dependent on current and past connectivity states

Past Connectivity	***Current Connectivity***	***The State of CRISP***
Unknown	*OFF*	*Emulate*
OFF	*ON*	*Reintegrate*
OFF	*OFF*	*Emulate*
ON	*ON*	*Harvest*
ON	*OFF*	*Emulate*

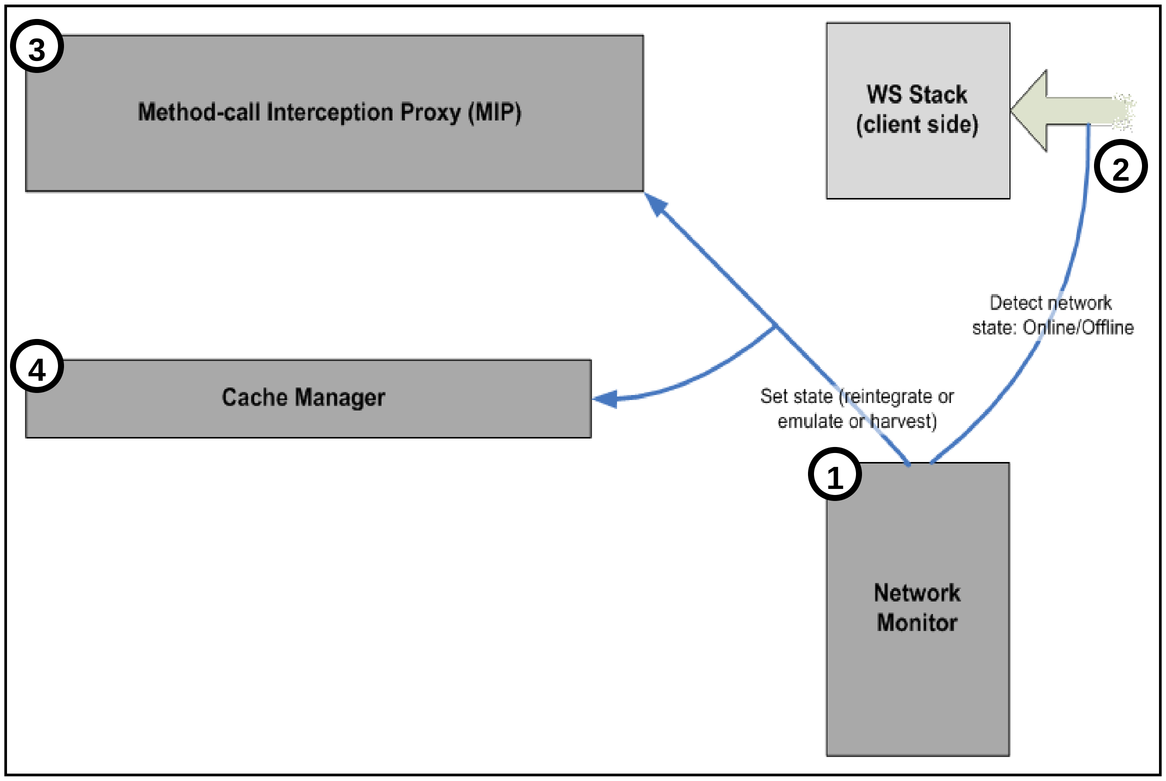

3
Method-call Interception Proxy (MIP)
WS Stack (client side)
2
Detect network state: Online/Offline
4
Cache Manager
Set state (reintegrate or emulate or harvest)
1
Network Monitor

Figure 3-5. The network monitor (1) detects the network state (2) and influences theMIP (3), and the cachemanger (4)

The Cache Manager

The cache manager [Figure 3-6 (2)] controls all cache operations. These operations are initiated by the MIP [Figure 3-6 (1)]. The cache manager interfaces with the CSD [Figure 3-6 (3)] preliminary to making cache decisions (e.g. add to cache, add to log).

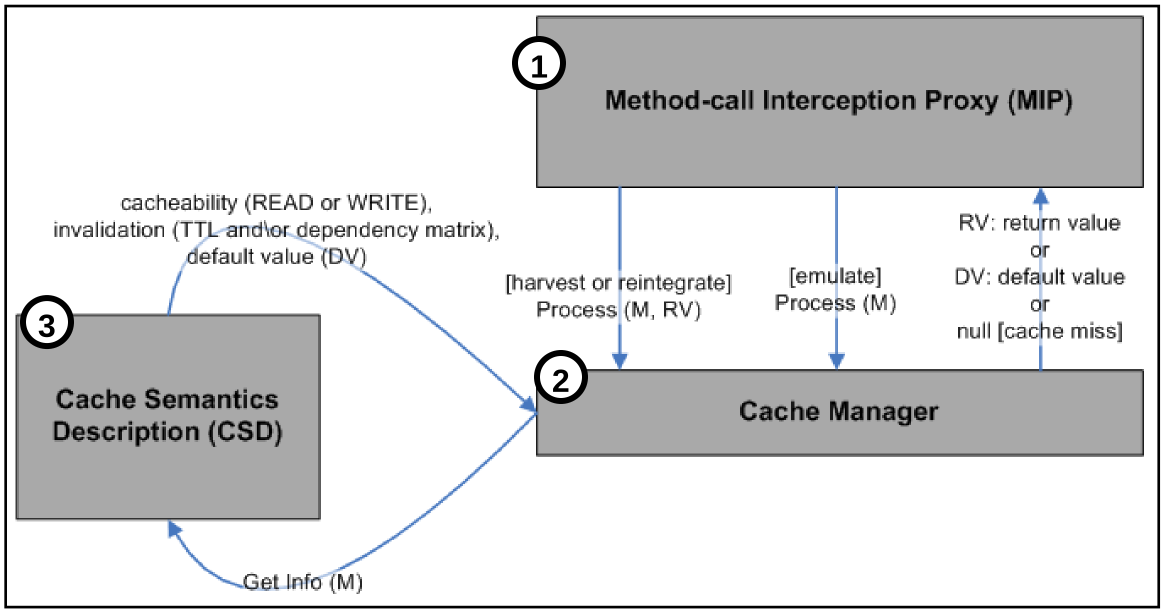


Figure 3-6. On every request from the MIP (1); the cache manager (2) queries the WS cache model (3)

The cache manager queries the CSD, if a request is identified as a READ (e.g. *GetRecord*) then the cache manager queries the hashtable [Figure 3-7 (2)]. If the hashtable contains a return value of the request, then the result is forwarded back to the MIP. However, if no local return value is found (a cache miss) then this is handled depending on the state of CRISP. In the emulate state, the cache manager returns a default value (DV), if known, and adds the request to the READ queue [Figure 3-7 (3)]. However, in the harvest state, the cache manager returns a null which indicates that the request should be processed remotely. In turn, the MIP forwards a pair of method invocation/return back to the cache manager after the request is processed by the WS.

Alternatively, the CSD can identify a request as a WRITE (e.g. *AddRecord*). The cache manager proceeds depending on the state of CRISP. If the state is emulate, then the cache manager adds the request to the WRITE queue [Figure 3-7 (4)]. However, if the state is harvest, then the cache manager returns a null to the MIP, which in turn sends the request to the lower levels of the WS stack, with the goal of processing by the WS.

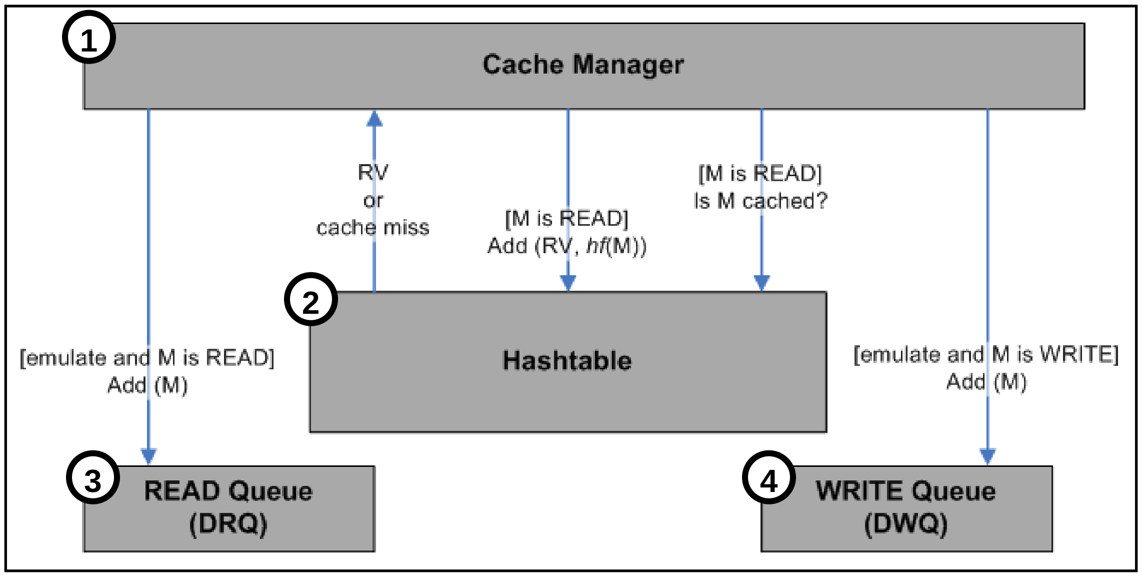


Figure 3-7. The cache manager (1) queries the hashtable (2) if a request is cached, depending on the result and the current state: the request is added to the READ queue (3) or the WRITE queue (4).

The Cache

The cache is composed of a hashtable and two queues; the READ queue (DRQ) and the WRITE queue (DWQ) [Figure 3-7 (2, 3, 4)]. All operations are performed by the cache manager component [Figure 3-7 (1)]. The queues are populated in the *emulate* state, and are emptied when the state becomes *reintegrate*.

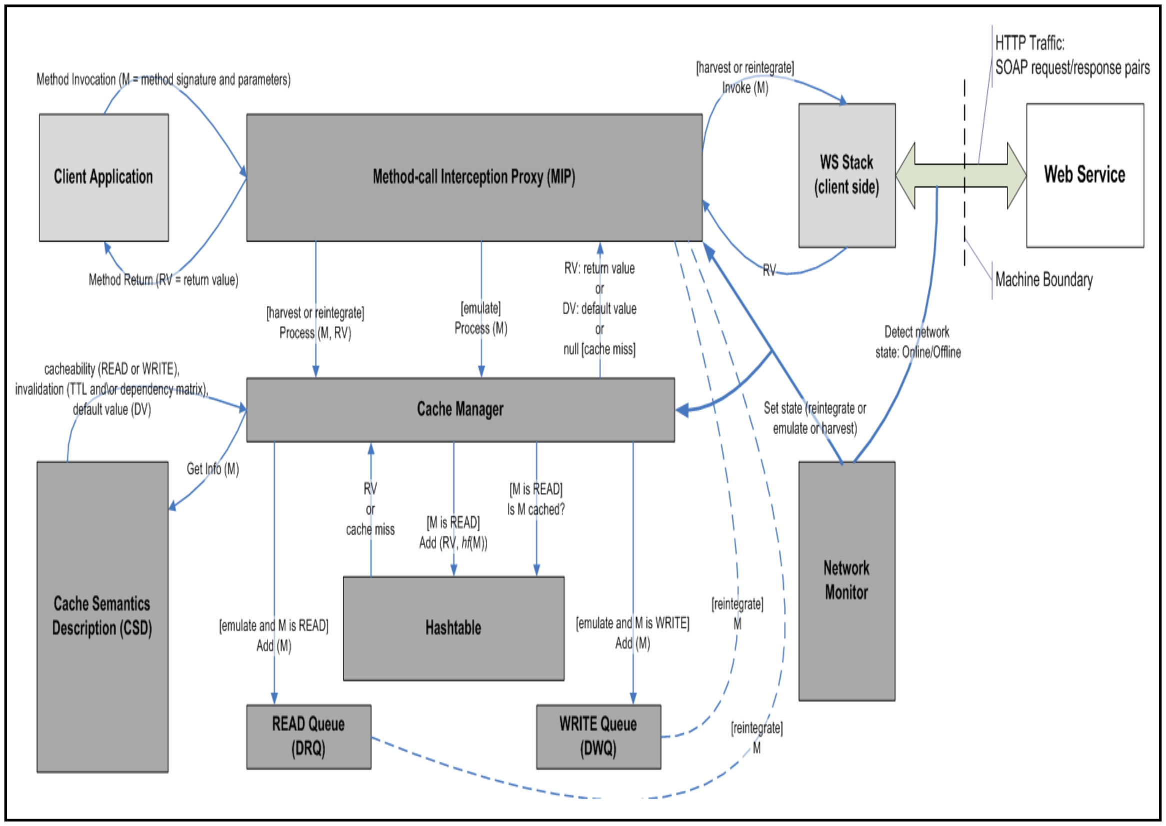
The hashtable uses a function *hf* parameterized by the WS location, method name, and any parameters specified by the CSD. The function *hf* is a hashfunction used for storing a response, and mapping stored responses to new requests.

The DRQ and DWQ contain READ and WRITE requests, respectively. The DRQ guarantees uniqueness. The guarantee of uniqueness is an optimization, permitted because the outcome of N identical READs is equivalent to the outcome of the first READ. The same guarantee, however, is not implemented by the DWQ. The DWQ permits duplicates because the state of the WS is dependent on the order, and number of WRITEs.

Consistency Maintenance

Consistency maintenance in CRISP does not require WS cooperation. Hence, implicit invalidation (CAO 98, NOB 97) is used. The invalidation condition of a record is dependent on the CSD's specification. The CSD can specify a response's expiry on a threshold of time (e.g. 1 hour). Alternatively, the CSD can specify a directed state graph, which results in the invalidation of a cached READ (e.g. *GetRecord*) cascaded by the execution of a WRITE (e.g. *ModifyRecord*).

Upon reestablishment of connectivity, the network monitor triggers the reintegration phase (inspired by Coda (NOB 97)). The DWQ buffers state-altering methods invoked during disconnectivity, in FIFO order. The reintegration phase commences by invoking the invocations stored in the DWQ, followed by the invocations stored in the DRQ. Depending on the cache policy, the write back of the queues is either preemptive of new requests, or is performed in the background (a.k.a. trickle reintegration).



Method Invocation (M = method signature and parameters)
Client Application
Method-call Interception Proxy (MIP)
Method Return (RV = return value)
[harvest or reintegrate] Invoke (M)
WS Stack (client side)
Web Service
HTTP Traffic: SOAP request/response pairs
Machine Boundary
RV
[harvest or reintegrate] Process (M, RV)
[emulate] Process (M)
RV: return value or DV: default value or null [cache miss]
Detect network state: Online/Offline
cacheability (READ or WRITE), invalidation (TTL and/or dependency matrix), default value (DV)
Cache Manager
Set state (reintegrate or emulate or harvest)
Get Info (M)
Cache Semantics Description (CSD)
RV or cache miss
[M is READ] Add (RV, hf(M))
[M is READ] Is M cached?
Hashtable
Network Monitor
[emulate and M is READ] Add (M)
[emulate and M is WRITE] Add (M)
[reintegrate] M
READ Queue (DRQ)
WRITE Queue (DWQ)
[reintegrate] M

Figure 3-8. Detailed architecture diagram of CRISP

Cache Model Specification

CRISP requires the specification of a WS cache model. This model is specified as an XML document named the *cache semantics description (CSD)*. The following sections describe the fundamental model elements.

The cacheability element

The cacheability semantic is defined when methods are classified as a READ and/or a WRITE. Cacheability is assumed for READs (e.g. *GetRecord*), and WRITEs (e.g. *AddRecord*) are logged for write back. A WS method is marked as a READ if it is state-dependent. A state dependency is assumed if the method defines a return value [Table 3-2]. On the other hand, a WS method is marked as a WRITE if it is state-altering. State alteration is assumed if the method does not define return a value [Table 3-2].

Table 3-2. Identification of method cacheability in the CSD

Method signature	Return Value	CSD Marking
datatype *GetRecord*(parameters)	yes	READ
AddRecord(parameters)	no	WRITE
ModifyRecord(parameters)	no	WRITE

The dependency matrix element

The dependency matrix is used for specifying the functional dependencies of methods. The dependencies are used for consistency maintenance in CRISP, they encode the relationship [Table 3-3] between WS methods and internal resources (e.g. database tables). The matrix is a directed relationship graph between N methods and X resources.

Table 3-3. A specification of a simple dependency matrix element in the CSD

Method	Dependent on Method
GetRecord	*ModifyRecord*
AddRecord	
ModifyRecord	

In the above table, the *GetRecord* method is singularly dependent on ModifyRecord. The dependency, however, does not specify a shared parameter as a condition for dependency. Hence, the complexity of the dependency matrix is low and all cached instances of *GetRecord* are removed from cache after any execution of *ModifyRecord*. This matrix assumed that there is a single resource (e.g. database table) which is shared between GetRecord and ModifyRecord. However, in reality, many resources are shared (e.g. database rows) and only one resource at a time is relevant. The relevant resource is identified by a shared parameter (e.g. record identifier).

The WS methods use X resources, and each resource is denoted by R. A method either READs or WRITEs a resource. The dependency matrix [Table 3-4] encodes a directed relationship graph [Figure 3-9]. The result is that a cached method M_i (state-dependent on R_α) is invalidated/refreshed when M_j (state-altering of R_α) is invoked.

Table 3-4. A specification of a more complex dependency matrix element in the CSD

Method	Dependent on Method
GetRecord(R_α) M_1	*ModifyRecord*(R_α) M_2
AddRecord M_3	
ModifyRecord M_2	

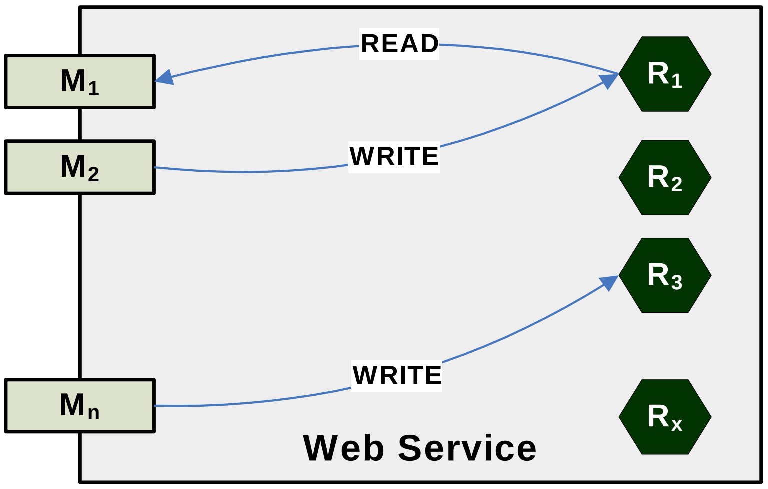


Figure 3-9. Directed state graph between N methods (denoted by M) and X resources (denoted by R)

The default values element

Terry at al. (TER 03) proposed the specification of *default values* for WS methods. A default value shields the application logic from the effects of a disconnected cache miss. Default values are defined for READs [Table 3-5], at the model design time, and are used for initializing the CSD. In this work, the use of default values is termed *partial cache hits*.

Table 3-5. A specification of simple default values in the CSD

Method	Default Value
GetRecord	*“temporarily unavailable”*

Summary

The design of a client-side cache and a WS cache model are the subject of this chapter. The client-side cache, implemented on Windows Mobile is named CRISP, and is used for experimentation. CRISP is located within the client-side WS stack, it intercepts method invocations and method returns and performs an action. One of three actions is possible; a method invocation is sent to the WS, stored in a local log, or answered from cache. The action performed depends on the state of CRISP and on the WS cache model. Hence, the operation of CRISP requires a specification of the WS cache model.

The WS cache model consists of three elements; cacheability, dependency matrix, and default value elements. The model is specified as an XML document named the cache semantics description (CSD). A valid CSD specification requires only the specification of the cacheability element. Cacheability is specified by assigning READ/WRITE semantics to WS methods. The complexity of the CSD can then be increased by including the dependency matrix. The dependency matrix represents the functional dependencies of WS methods.

The operations of CRISP are transparent to both client applications and remote WS. CRISP maintains three states; harvest, emulate, and reintegrate. The state of CRISP (e.g. emulate) is identified by the relationship between current connectivity and past connectivity (e.g. current=OFF and past=ON/OFF). The role of connectivity detection and state maintenance is internal to CRISP. CRISP detects the state of OFF connectivity, buffers WRITEs, and answers READs locally. Once connectivity is regained, then CRISP performs consistency maintenance, driven by the cache semantics description. The design of CRISP, the cache semantics, and the location of the framework are the three factors that enable transparent adaptation of the WS client to intermittent connectivity.

CHAPTER 4
EXPERIMENTATION

The evaluation of this research is carried out by three sets of experiments, alphabetically labelled A, B and C. In each set, experiments are assigned a goal and one or more workload(s), metric(s), cache model(s), connectivity pattern(s), and cache policy. For each experiment, a minimum of one setting from each of the following settings is allocated: one WS, five synthetic workloads, three cache models, five network connectivity patterns.

The choice of using five workloads is to have workloads containing a high to low ratio of READ and WRITE operations. Five workloads permit having workloads containing from 100% READs to 20% READs, and from 0% WRITEs to 80% WRITEs, at 20% steps. The five network connectivity patterns, on the other hand, capture 100% connectivity down to 20% connectivity, at a 20% reduction of connected times in each pattern.

In the first experimental set A, the cost of using the framework vs the no-framework scenario is compared. Additionally, the cache efficiency of a simple cache model is measured. The comparison is performed in strong and weak connectivity settings for the framework scenario. Experimental set B evaluates cache efficiency in relation to model complexity, using three cache models. The third experimental set (C), measures the influence of workload characteristics, network connectivity pattern on cache efficiency and on client cost.

The first half of this chapter (0) details the experimental setup, and the second half (0) details the three experimental sets.

Experimental Setup

The Web Service

The experimental WS implements methods supporting a public discussion forum. The use of a public discussion forum is seen as a realistic medium for mobile user learning. Mobile users will initiate new discussions (by asking questions), contribute to new discussions by replying to existing topics and modify old posts. The WS supports common activities of public discussion forums by implementing a set of five operations, detailed by the following list of operations.

WS Methods:

The WS implements the following five methods, in no particular order:

- Get the number of messages in a forum (READ)
- List the messages of a forum (READ)
- Read a message (READ)
- Add a message (WRITE)
- Modify a message (WRITE)

WS Cache Models

Three models are defined (see Table 4-1), the first model defines cacheability semantics and no consistency maintenance semantics are defined. In the second model, simple semantics supporting coarse consistency maintenance is defined. In the third model, detailed consistency maintenance is enabled by using a refined dependency matrix. The WS is defined to have N methods, maintaining X resources.

Cache Model 1: Optimistic Model

The model specifies a simple cacheability element for N methods. The cacheability element defines READ, WRITE semantics for each method. The model does not define any semantics for consistency maintenance. As a result the model is optimistic on the validity of the cached READs.

Cache Model 2: Pessimistic Model

In addition to the specification of simple cacheability, the model also specifies an additional element supporting consistency maintenance. The model specifies a simple dependency graph, representing the relationship between N methods and X resources. However, the X resources are approximated by a single resource R [Figure 4-1], as method arguments are not considered. The model is pessimistic because consistency maintenance is managed by the coarse granularity of the dependency graph. The approximation of resources by a singular resource R leads to the invalidation of all READs, on every execution of a WRITE.

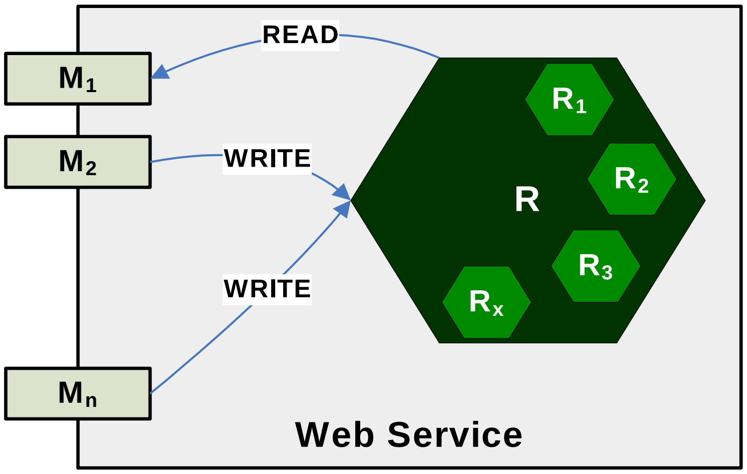


Figure 4-1. The relationship of N methods (denoted by M) and a single resource (denoted by R), the original X resources (R_1… R_x) are approximated by R

Cache Model 3: Resource-Aware Model

This model is a refinement of model 2. The resource relationships are specified for all X resources [Figure 4-2]. The granularity of the relationships is improved as the method arguments are considered for identifying resource accesses.

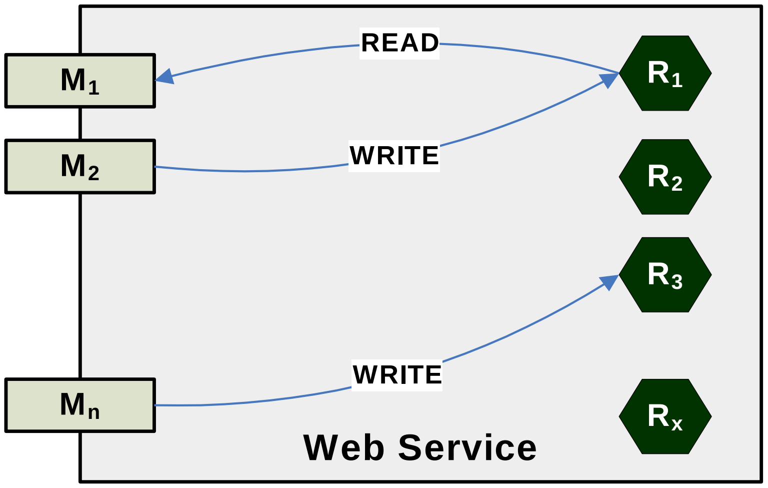


Figure 4-2. The relationship of N methods (denoted by M) and X resources (denoted by R)

Table 4-1. The three Cache Models

Model	***Cacheability***	***Resource Relationships***	***Consistency Maintenance***
1. Optimistic Model	yes	0 resources	none
2. Pessimistic Model	yes	1 resource	simple
3. Resource-Aware Model	yes	X resources	complex

Metrics

The metrics are categorized by the nature of their measurement (e.g. cost, efficiency). In the first category are the metrics measuring cache efficiency. The second category is a set of metrics measuring network costs. In the third category, the metrics measure the costs expended by the WS client process (e.g. CPU Time).

Cache Efficiency

- Cache Hits: the number of locally found responses
- Consistent Cache Hits: the number of consistent locally found responses
- Cache Efficiency: the percentage of cache hits which are also consistent records accesses. Efficiency = #Consistent Cache Hits ÷ #Cache Hits

Network Cost

- Network Bytes - Inbound: the total byte size of inbound responses
- Network Bytes - Outbound/Delayed: the total byte size of outbound requests, grouped by connectivity state.

Client Cost

- Request Latency – Time: the total time spent between a method call and receiving the method return
- CPU – Time: the time spent by the CPU while processing a method call/return, this metric includes the overhead time of framework operation.
- Memory – Bytes: the memory size of the client process, the size includes the size of the cache, cache content, and the process's execution stack and heap.

The client is instrumented to collect the metrics while it is subjected to a set of scenarios (e.g. Workloads). The client executes dependently on a request/connectivity pattern, and the metrics are collected after every request.

Workloads

The experimental workloads are a set of synthetic request patterns. A synthetic workload is a set of WS requests following the Zipf distribution. The Zipf distribution is used for selecting the record ID. The requests are characterized by the ratio X (READ operations) to Y (WRITE operations), such that a workload is composed of X% READs and (100-X)% WRITEs. For all workloads, optimistic concurrency is assumed and an experimental run assumes no write-write conflict.

Synthetic Workloads

P1; 100% READs, 0% WRITEs

P2: 80% READs, 20% WRITEs

P3: 60% READs, 40% WRITEs

P4: 40% READs, 60% WRITEs

P5: 20% READs, 80% WRITEs

Network Connectivity Patterns

A fault generator is responsible for emulating intermittent connectivity. The connectivity is fluctuated between Online and Offline, at uniform random intervals and for uniform periods of time. An 80% Connectivity pattern means that the client is able to access the WS in 80% of the requests.

C1: 100% Connectivity (strong connectivity)

C2: 80% Connectivity (weak connectivity)

C3: 60% Connectivity (weak connectivity)

C4: 40% Connectivity (weak connectivity)

C5: 20% Connectivity (weak connectivity)

Cache Policies

A READ policy (RP 1) and two WRITE polices (WP 1-2) are enabled by the framework. Unless stated, the experiments use the default READ policy RP 1 and the default WRITE policy WP 1. The least recently used (LRU) policy is used as the default replacement policy. In these experiments, the cache size is set to a large value, and as a result the replacement policy will not be executed. The replacement policy is only needed when the size/number of cached responses is close to the maximum capacity of the cache.

Invalidation of cached elements is performed by implicit invalidation. A cached response expires by an age value, or when a resource-sharing WRITE is invoked. The WS cache model [Figure 0] details the invalidation conditions and parameters per WS method.

Replacement Policies:

Least Recently Used LRU

This policy has been shown to support short-term temporal locality in web caches (e.g. (MAH 00, WAN 99)) and caches in general (e.g. KIS 92, CAO 95, FRO 98)). The policy removes the oldest cache record first. The goal is that the cache contains primarily records which are in demand.

READ Policies:

RP 1: Default

In the default policy CRISP exploits local responses, regardless of connectivity state. As a result, the latency of requests is reduced at the cost of a decrease in consistent record accesses. CRISP can be configured to not exploit temporal locality, and instead to answer requests from cache only when network connectivity is lost.

In addition, by default, CRISP implements an optimization suggested by Terry et al. (TER 03) transforming a disconnected cache miss to a partial cache hit (read-default). In addition to masking a cache miss by a partial cache hit (read-default), CRISP implements a simple optimization of the READ queue (DRQ). In response to a disconnected cache miss, the request is also queued for delay reading once connectivity is regained.

WRITE Polices:

WP 1: Background/Trickle write back

Trickle write back during reintegration has been shown (FRO 98, CHE 95) to result in a better overall read throughput, without a significant decrease in the time of return to a consistent state.

WP 2: Preemptive write back

It has been shown (FRO 98, CHE 95) that preemptive write back improves the consistency of the cache at a faster rate than is obtained by delayed write back policy. The improvement of consistency is the result of write preference, at the expense of a decrease in read throughput.

Execution Environment

The experiments are executed within virtual machines [see Figure 4-3] in order to control the environment. The virtual machines are configured to contain 512MB Ram, and a default Pentium III-class processing unit. An instrumented WS client uses a WS, given an experiment configuration. A configuration consists of a set of workloads, connectivity patterns, and a cache model.

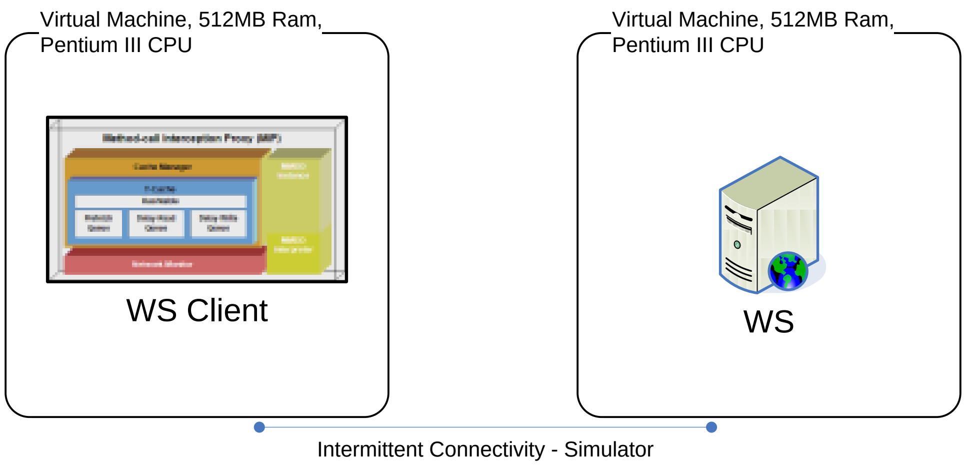

Virtual Machine, 512MB Ram, Pentium III CPU
Method-call Interception Proxy (MIP)
Cache Manager
WS Client
Virtual Machine, 512MB Ram, Pentium III CPU
WS
Intermittent Connectivity - Simulator

Figure 4-3. The execution environment

Experiments

Experimental set A

The goal of these first experiments is to demonstrate the benefit of a WS cache framework when compared to the no-framework scenario. The experiments measure the achieved cache efficiency of the framework scenario and then compare the client cost of using the framework to the client cost of the no-framework scenario.

The framework scenario uses an Optimistic Model, as it is the simplest of all models. The comparison is made in strong connectivity (the first experiment), and is repeated again in weak connectivity (the second experiment). The specification of a WS cache model defines the cacheability of WS methods, and enables utility of the WS client during intermittent connectivity.

Experiment 1: No cache model vs. Optimistic Cache Model

The experiment is executed in the strongly connected setting (network connectivity pattern C1). The goal of the experiment is to measure the client cost of a WS client using the framework, and compare it to the client cost of the WS client without using the framework. The metrics collected are the client cost metrics. The workloads of this experiment are P1-P2 and only the default cache policy is used.

Experiment 2: Optimistic Cache Model in weak connectivity

The experiment subjects the framework scenario to weak connectivity (connectivity pattern C2). The goal of the experiment is to highlight the benefit of using the framework during intermittent connectivity. In the weakly connected setting, using a cache model enables utility of a WS client, and cache efficiency is measured. The client cost is also measured and compared to that of the strongly connected setting. The workloads of this experiment are P1-P2 and only the default cache policy is used.

Experimental set B

Three experiments evaluate the effect of model complexity on cache efficiency. It is expected that as model complexity increases, that cache efficiency will also increase. All the experiments are executed under a weak connectivity constant using connectivity pattern C2. Each experiment is executed using a workload containing 20% WRITE, and 80% READ operations.

Experiment 1: Cache efficiency of the Optimistic Cache Model

The experiment executes an Optimistic Model (cache model 1) under weak connectivity (C2) and synthetic workload (P2). The goal of the experiment is to measure the cache efficiency in a baseline setting. The metrics collected are the cache efficiency metrics.

Experiment 2: Cache efficiency of the Pessimistic cache model

The experiment executes a Pessimistic Model (cache model 2) under weak connectivity (C2) and synthetic workload (P2). The goal of the experiment is to identify the observed cache efficiency in comparison to the results of the first experiment. The metrics collected are the cache efficiency metrics. The default cache policy is used.

Experiment 3: Cache efficiency of the Resource-Aware cache model

The experiment executes a Resource-Aware model (cache model 3) under weak connectivity (C2) and synthetic workload (P2). The goal of the experiment is to identify the observed cache efficiency in comparison to the results of the first two experiments. The metrics collected are the cache efficiency metrics. The default cache policy is used.

Experimental set C

The experiments measure the effect of two factors on cache efficiency and on the client-side cost. The goal of these experiments is to measure the influence of workloads, and connectivity pattern using three cache models. It is expected that the costs at the client are composed of the cost of model processing, and the cost of framework operation.

The first experiment evaluates a baseline setting, while the following two experiments evaluate the influence of one selected factor at a time.

Experiment 1: Baseline cache efficiency and client cost of cache models

Three models (cache model 1-3) are examined using workload P2 and network connectivity pattern C2. The metrics collected are the client cost metrics, the network cost metrics, and the cache efficiency metrics. The goal of the experiment is to evaluate a baseline setting.

Experiment 2: Influence of workload on cache efficiency and client cost

Three model (cache model 1-3) are examined using workloads P3-P5 and network connectivity pattern C2. The goal of the experiment is to evaluate the influence of a workload on cache efficiency and cost.

Experiment 3: Influence of connectivity on cache efficiency and client cost

Three models (cache model 1-3) are examined using workload P2 and network connectivity patterns C3-C5. The metrics collected are the cache efficiency metrics, the client process metrics, and the network cost metrics. The goal of the experiment is to evaluate the influence of network connectivity on cost and on cache efficiency. The influence of network connectivity is compared to that of workloads.

CHAPTER 5
RESULTS

Experiment Set A

The experiments use an Optimistic Cache Model to demonstrate the benefit of the framework. The experiments measure the achieved cache efficiency and compare the client cost of using the framework to a no-framework scenario.

The experiments use two workloads, the first contains 100% READ requests and no WRITE requests, while the second contains 20% WRITE, and 80% READ requests. Cache efficiency is expected to decrease with the introduction of WRITE requests, as they result in the implicit invalidation of some cached responses. It is necessary to execute the no-framework scenario in strong connectivity, as only the utility of the framework permits the use of a WS client during intermittent connectivity.

Data that show CPU utilization (%), or Latency (time) are displayed using trend lines. A trend line is either a moving average, or a logarithmic average of the data points. A moving average is the average of K data points, per graph point (e.g. 50 per. Mov. Avg).

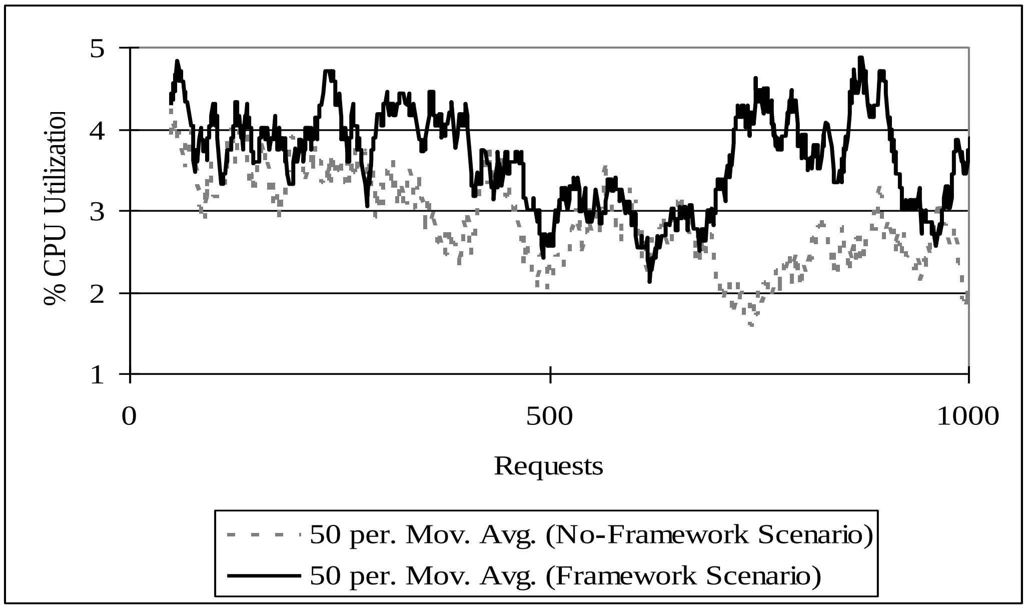


Figure 5-1. Execution cost of using the framework vs. the no-framework scenario

Observations and Discussion

The results [Figure 5-1] show the total CPU utilization when using CRISP, in comparison to the total CPU utilization without CRISP. The results show that CRISP, using an Optimistic Cache Model, during strong connectivity exhibit low overhead of an additional 50%. The CPU utilization of the no-framework scenario is between 2% and 4%, while the CPU utilization of the framework scenario is between %2 and 5%.

The execution cost (CPU) of a workload composed of 100% READ requests is above the execution cost of the no-framework scenario, however the difference is not significant. The same result is observed when using a workload containing 20% WRITE requests, however the cost is higher.

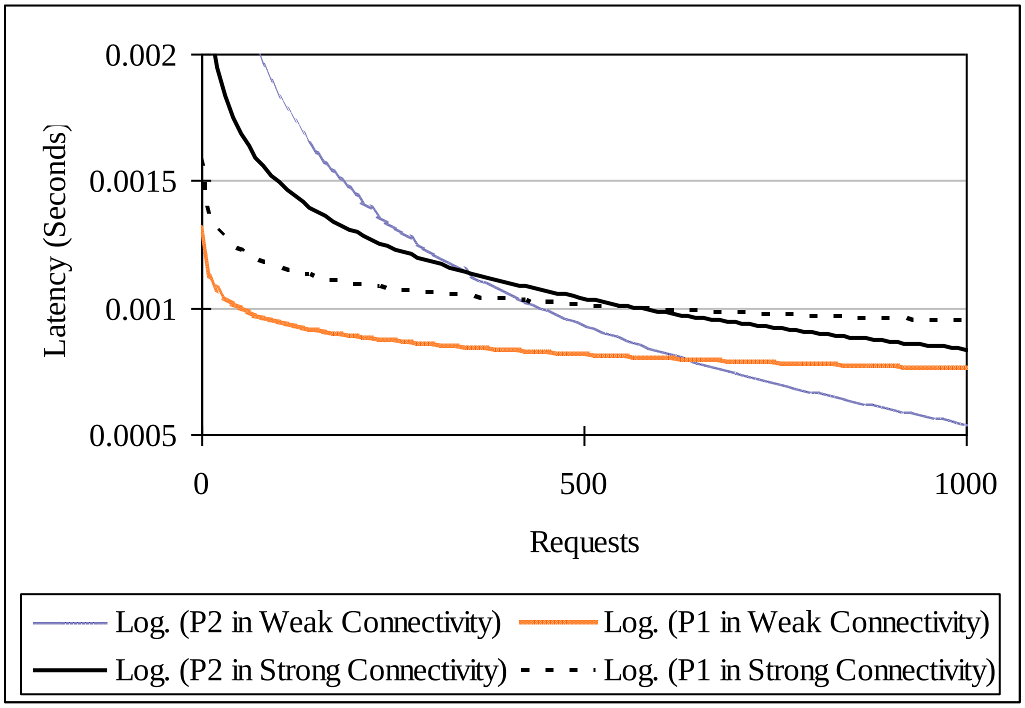


Figure 5-2. Request latency of framework, as Log of data, using workloads P1 and P2

In the strongly connected setting, a beneficiary effect of using the framework is an expected decrease in the latency of requests. It is observed [Figure 5-2] that request latencies are lower, in the strongly connected than in the weakly connected setting. The observation is true, for a short workload (e.g. 25% of P1, or 25% of P2). The observation is also true for a workload composed of READs only.

The trend, however, changes when using the full workloads [Figure 5-2]. The longterm latency of requests in a workload containing a mixture of READ/WRITE requests is better in the weakly connected setting. Additionally, the request latencies of the workload containing a mixture of READ/WRITE requests are better in the longterm than the request latencies of a READs only workload. This result can be attributed to the framework's overhead when processing READ requests.

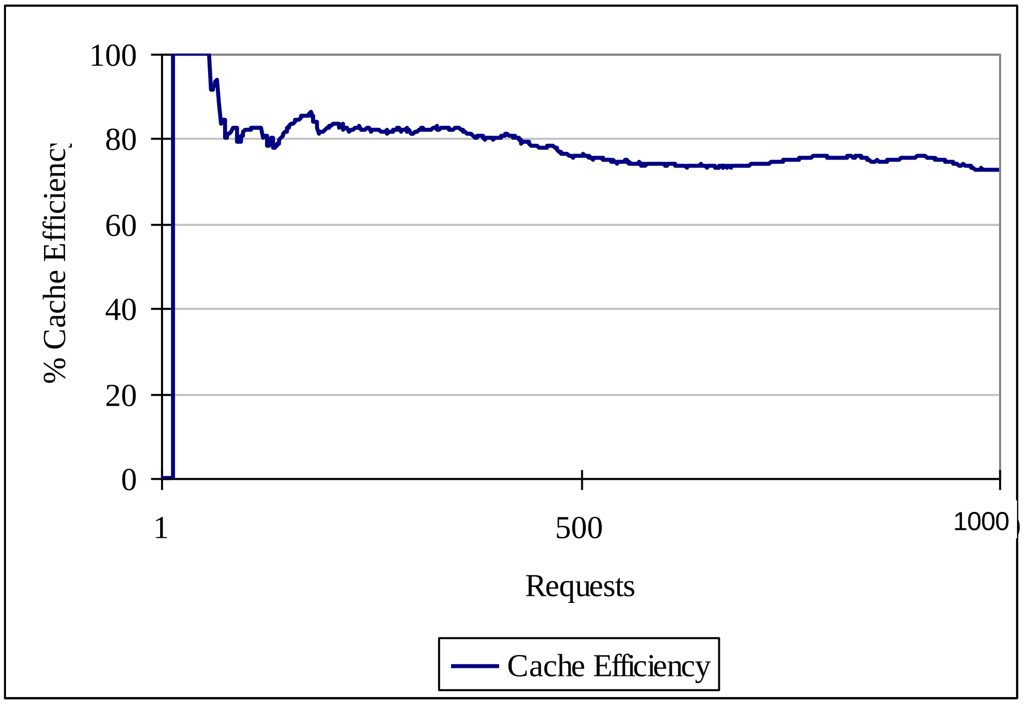


Figure 5-3. Cache efficiency using a mixed READ/WRITE workload (P2) during an 80% weakly connected setting

Using the framework during weak connectivity, and a workload composed of 80% READ requests, and 20% WRITE requests, the framework achieves a cache hit rate above 50% and cache efficiency above 70% [Figure 5-3].

A result of caching is the exploitation of temporal locality. The effect of locally answered READ requests is the reduction of execution and network costs for the mobile device. During the processing of P1 (workload composed of 100% READ requests) the execution cost linearly drops in time. However, the result is not observed when processing P2 (workload composed of 80% READ, and 20% WRITE requests), the processing cost increases as a result of the framework's overhead (e.g. reintegration).

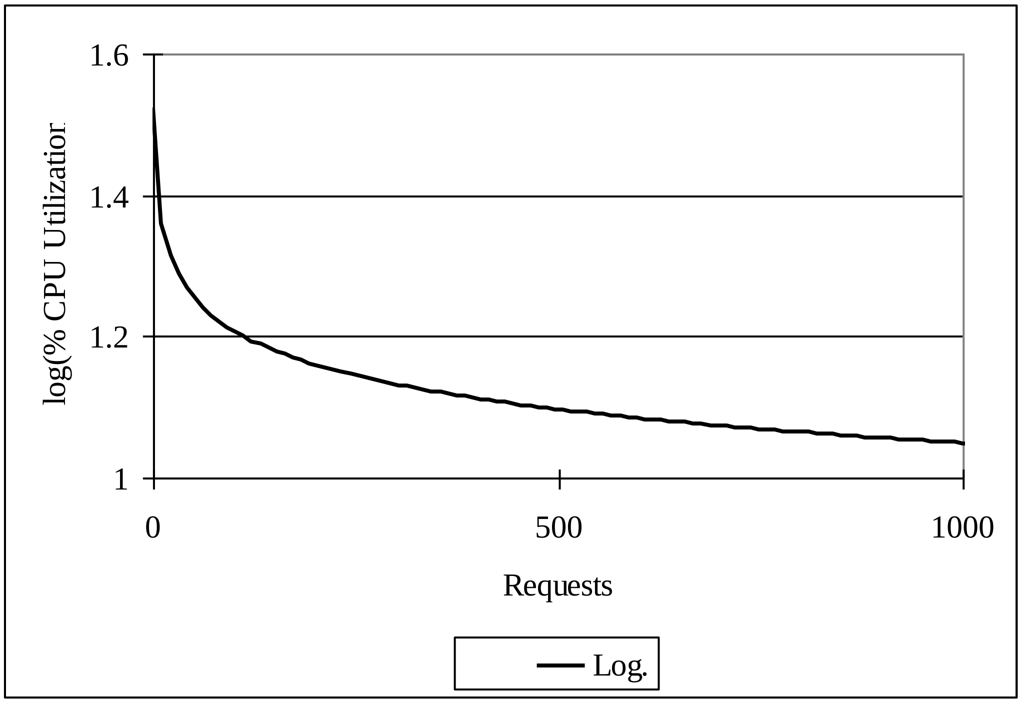


Figure 5-4. The linear reduction in processing cost when processing P1, shown as a log

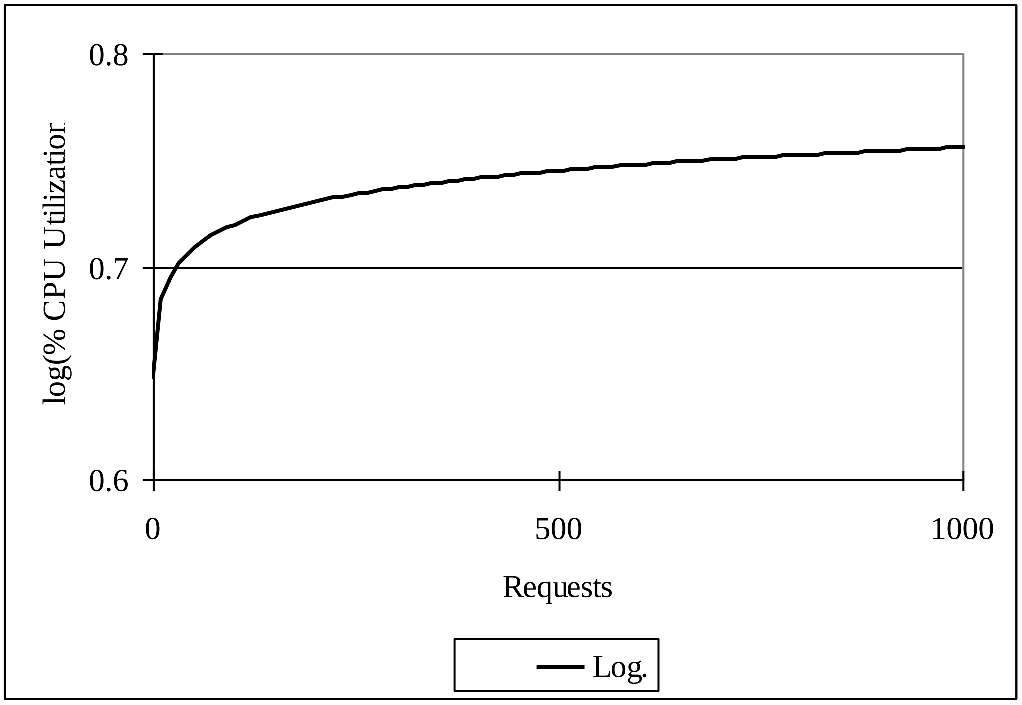


Figure 5-5. The linear increase of processing cost when processing P2, shown as a log

A reduction in network cost shows that local responses reduce network utilization.

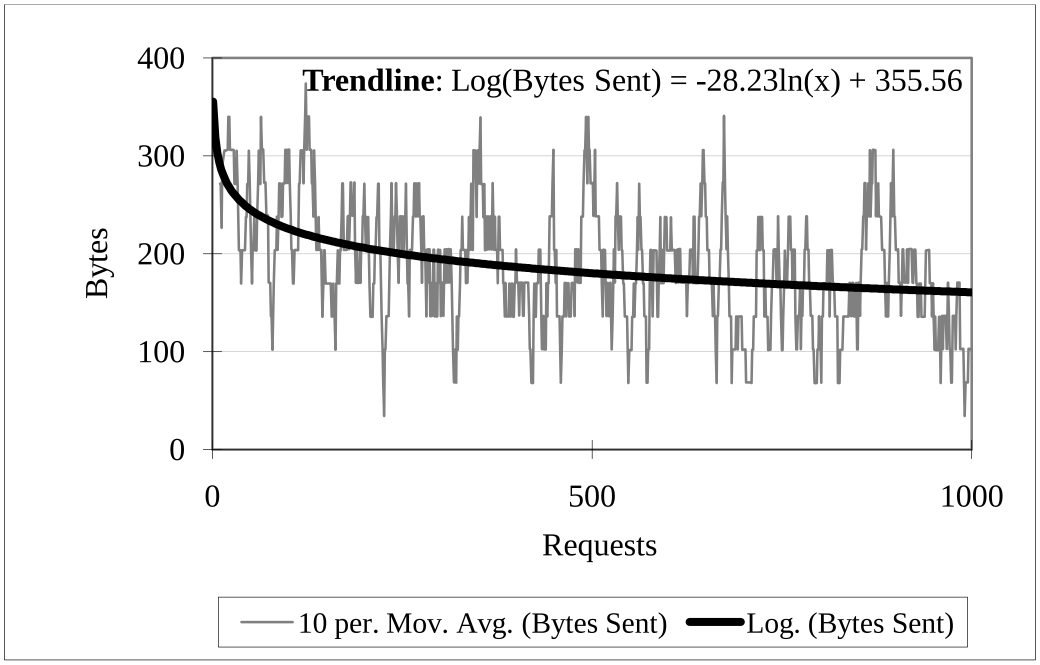


Figure 5-6. The reduction of network utilization when using the framework

An effect of weak connectivity is an increase of inconsistent record accesses, and an expected decrease of cache efficiency. The intermittent lack of connectivity prevents the return to a consistent state and slows consistency maintenance.

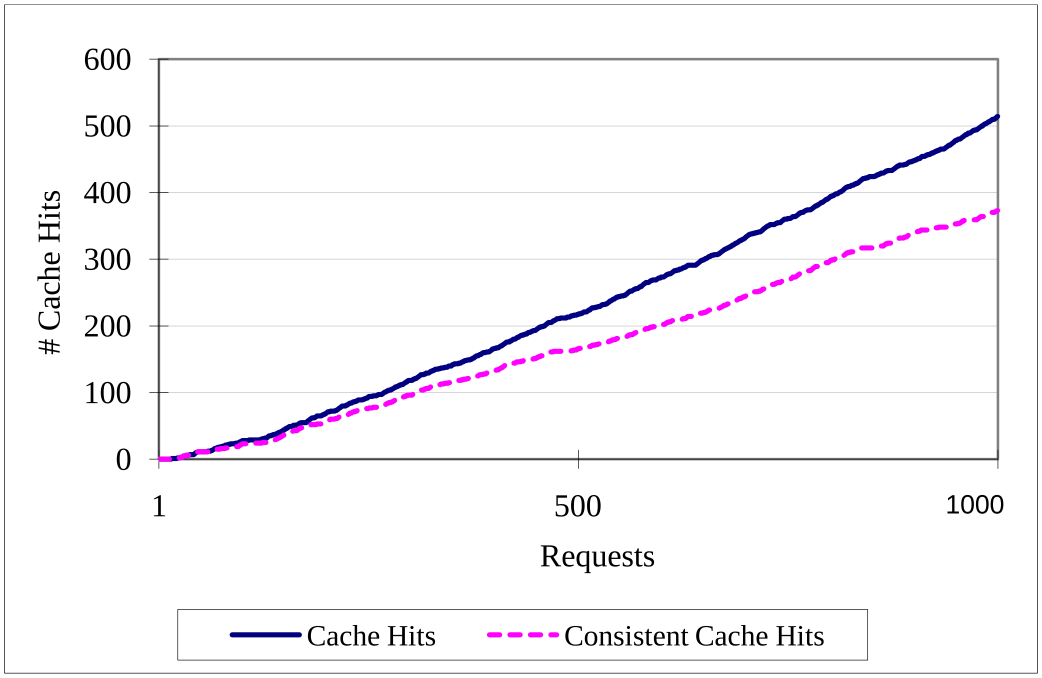


Figure 5-7. Cache hits and consistent cache hits using a mixed READ/WRITE workload (P2), weakly connected setting C2, and an Optimistic Cache Model

Caching is used with an implicit tolerance of stale records accesses. The achieved cache efficiency decreases over time [Figure 5-7] as cache consistency is not managed by the Optimistic Model. The underlying cause is that WRITE requests result in implicit invalidation of cached records, and the number of inconsistent cache hits increases over time as more WRITE requests are executed and consistency is not mainted.

Intermittent connectivity is a constant of mobility; however the degradation of cache efficiency over time is a negative result for using an Optimistic Cache Model. The primary benefit of using the framework is the utility of a WS client during intermittent connectivity. It is observed that the client cost is low, that the latency of requests is reduced while the achieved cache efficiency is acceptable.

Experiment Set B

The experiments use three WS cache models of increasing complexity. The first model contains cacheability semantics. The second cache model (Pessimistic Model) adds a dependency matrix, encoding the relationships between the WS method and resources. The matrix supports the invalidation/removal of stale READ records. However, the dependency matrix, in the Pessimistic Model, approximates all resources by a single resource R. The third cache model (Resource-Aware) refines the granularity of the dependency matrix by removing the approximation of resources.

The experiments are executed in a weakly connected setting, and use workload P2, containing 80% READs, and 20% WRITEs.

Observations and Discussion

The experiments show that when using an Optimistic Cache Model, the cache hit rate is 51.4%, and 72.6% of the cache hits are consistent record accesses [Figure 5-8]. During an extended period of weak connectivity, however, the Optimistic Cache Model is not sufficient for sustaining cache efficiency. The lack of consistency maintenance increases the number of stale record accesses over time.

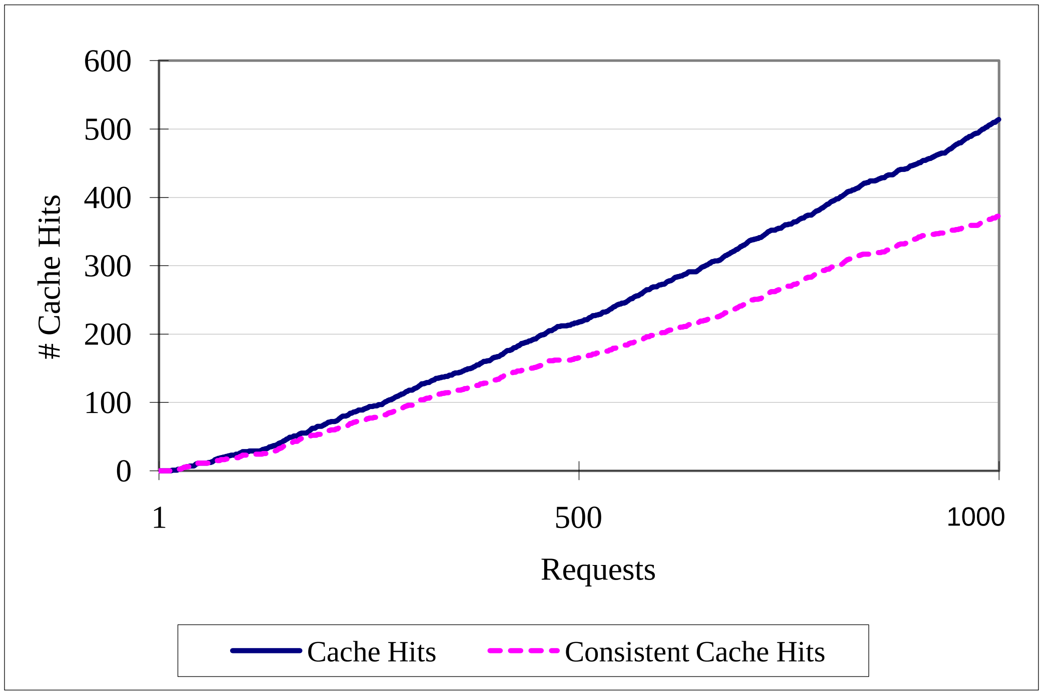


Figure 5-8. Breakdown of cache efficiency using Optimistic Cache Model

The lack of consistency maintenance in the Optimistic Cache Model can be overcome by introducing the dependency matrix element into the cache model.

We observe that using the Pessimistic Cache Model severely reduced the number of cache hits to below that of the Optimistic Cache Model (15.2% vs 51.4%) [see Figure 5-9]. The model's complexity was increased by introducing the dependency matrix element. However, the introduction of a coarse granularity dependency matrix resulted in lower cache efficiency (13.8% vs 72%). The coarse dependency matrix approximates all resources by a singular resource R. The approximation of resource accesses by R leads to a large set of false positives. False positives are cached records invalidated by a single WRITE request. A direct result is a smaller number of records remain in cache, increasing the number of cache misses and severely degrading cache efficiency.

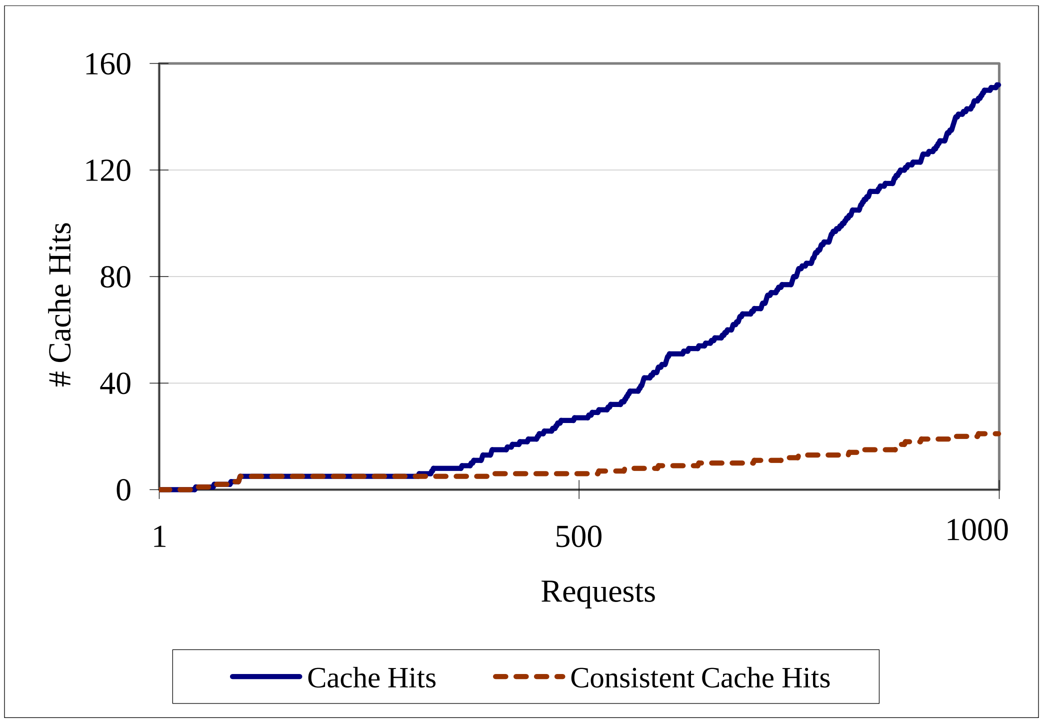


Figure 5-9. Breakdown of cache efficiency using Pessimistic Cache Model

The third experiment subjects the same workload and connectivity pattern to a Resource-Aware cache model. The model complexity is increased by refining the granularity of the dependency matrix. Instead of approximating x resources by R, all resources (R_1… R_x) are used. We observe the Resource-Aware cache model to successfully achieve cache efficiency above that of pervious models, at 99% [see Figure 5-11]. The result is achieved by the consistency maintenance enabled by the fine granularity dependency matrix. This result, however, is achieved at a slightly lower overall cache hit rate of 45.8%. Observe the following graph [Figure 5-10], it shows that the number of consistent cache hits nearly matches the number of overall cache hits. The result is that cache efficiency nears 100% as nearly all records accessed are in a consistent state.

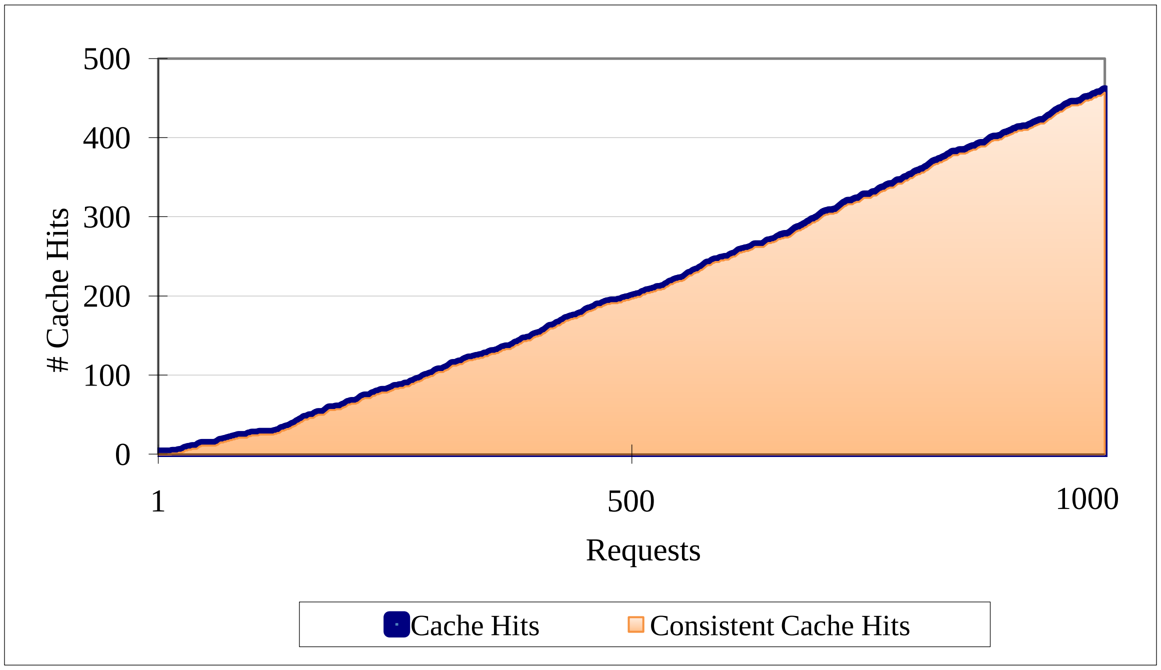


Figure 5-10. Cache hits are all nearly consistent, using the Resource-Aware cache model

The above result significantly improves the utility of the framework in mobile WS clients. High cache efficiency, as observed in the Resource-Aware cache model, enables the use of applications having low tolerance for stale records accesses. High cache efficiency (approaching 100%) means that nearly all record accesses are consistent. In turn, this satisfies a difficult requirement of applications that require consistent (or nearly consistent) record accesses.

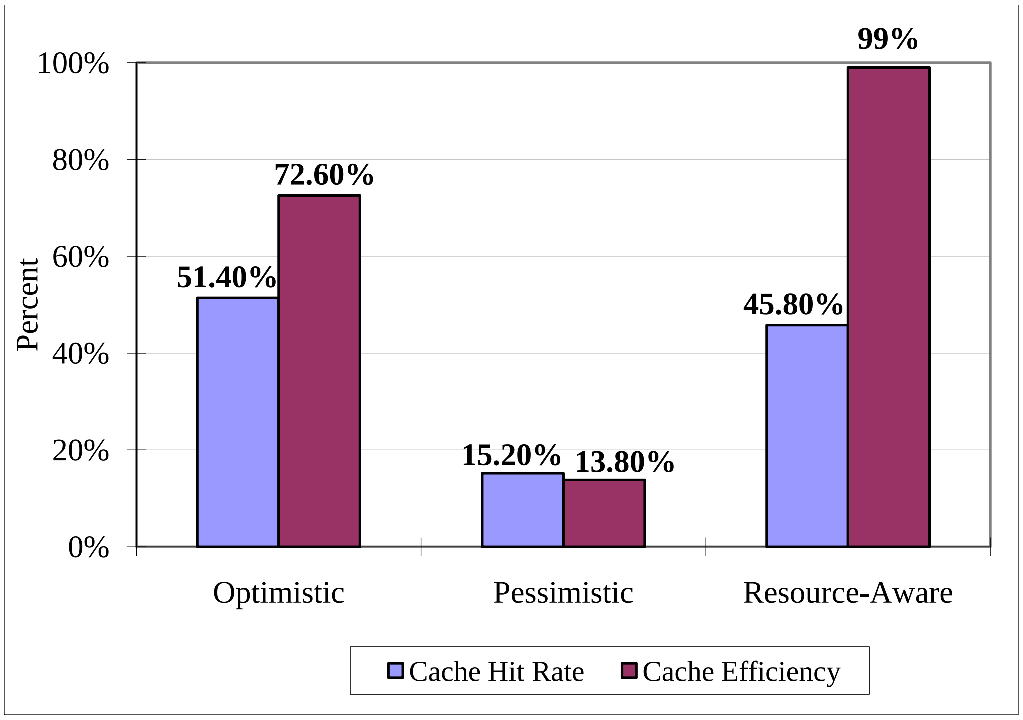


Figure 5-11. Cache hits, and cache efficiency, when using cache models of varying complexity

The results show [Figure 5-11] that an Optimistic Cache Model achieves cache efficiency above 70%. The results also show that increasing the cache complexity does not automatically increase cache efficiency. The Pessimistic Model produced significantly lower results than the optimistic and Resource-Aware models. The number of cache hits is highest when using the Optimistic Cache Model, however cache efficiency is highest when using the Resource-Aware cache model. We've also shown that it is possible to achieve a significant cache efficiency of 99% while maintaining the cache hit rate above 45%.

Experiment Set C

The experiments measure the effect of workloads and connectivity pattern on cache efficiency and on the cost at the client-side. The measurements are grouped by cache model complexity.

Observations and Discussions

Subjecting an Optimistic Cache Model to workloads P2-P5 proportionally decreases the achieved number of cache hits [Figure 5-12]. In the workloads P2-P5, the number of READ operations starts at 80% (in P2) and decreases by 20% for each workload (e.g. 60% in P3). The reduction in the number of READ operations is combined with a proportional increase of the number of WRITE operations. As a result, the number of cache hits decreases linearly with the decrease in number of READ operations in a workload.

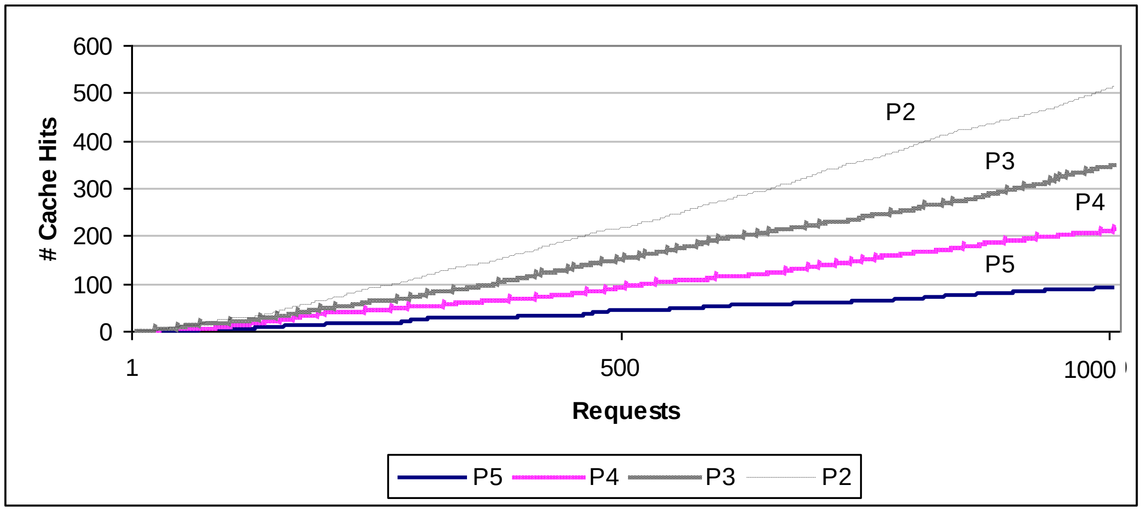


Figure 5-12. Effect of Workload on Cache Hits - Optimistic Model

Furthermore, the reduction in the number of cache hits, as the number of READ operations is reduced, is accompanied by a decrease in cache efficiency [Figure 5-13]. However, the decrease in efficiency is not at the same rate of cache hit reduction. Observe the curve representing the cache efficiency of P3 vs that of P2, then compare the separation of the curves with that of the efficiencies of P3, P4, and P5.

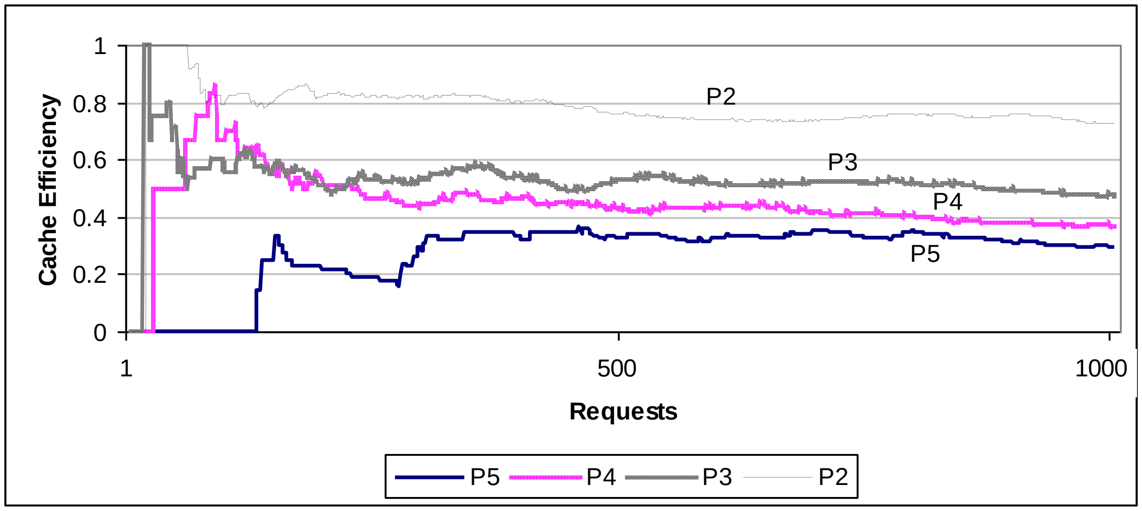


Figure 5-13. Effect of Workload on Cache Efficiency - Optimistic Model

Next, we observe the effect of varying workloads on the latency of requests [Figure 5-14], and on the processing cost [Figure 5-15] of the Optimistic Cache Model. Observe that request latency increases in the long term as the number of WRITEs increase. However the effect of workloads on the processing cost, in this model, is not significant. This result can be explained because the model complexity has a larger effect on processing cost, and request latency, than the effect of the distribution of READ/WRITE requests within a workload.

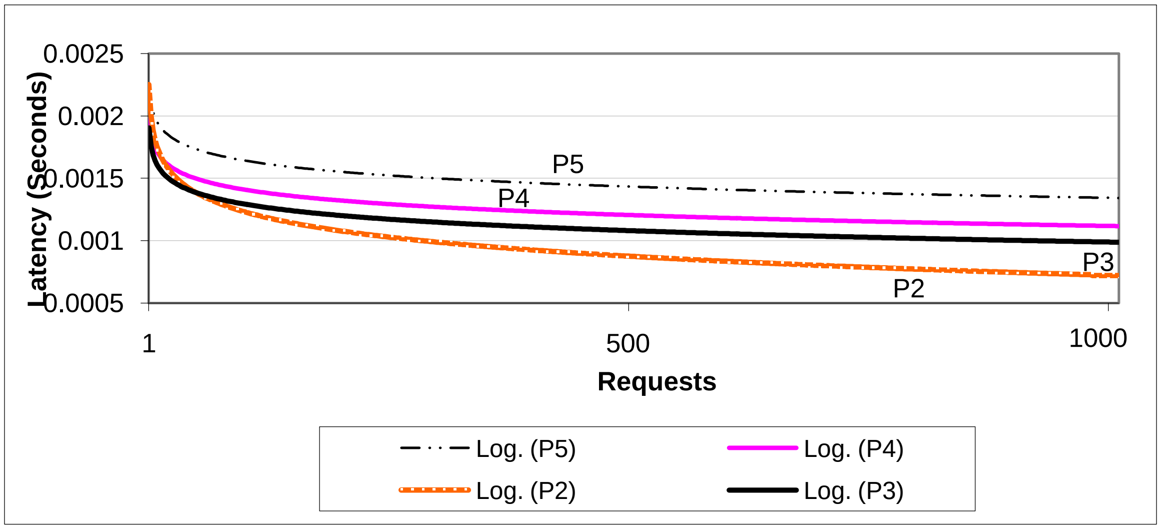


Figure 5-14. Effect of Workload on Request Latency – Optimistic Model

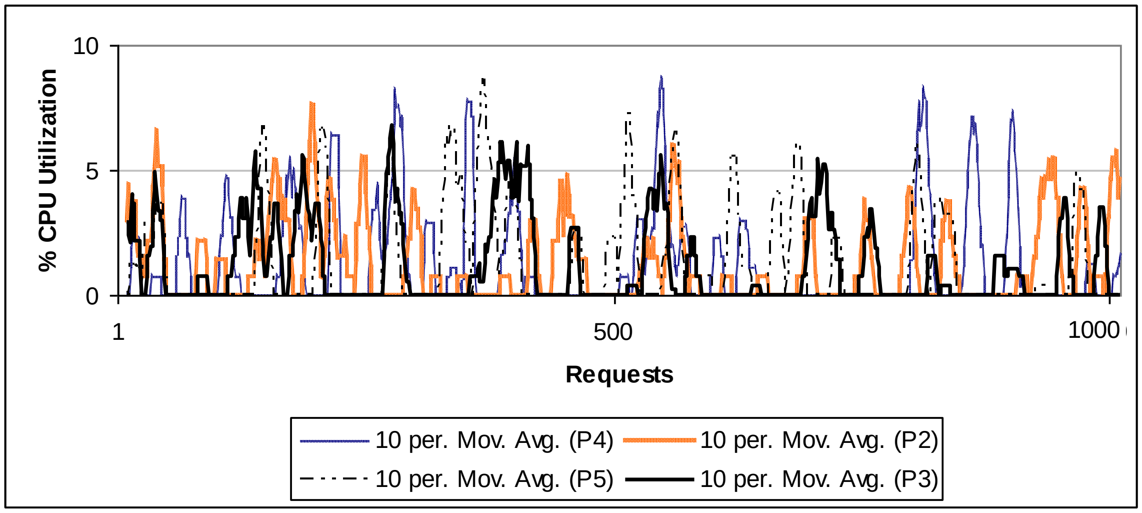


Figure 5-15. Effect of Workload on CPU Utilization on the client – Optimistic Model

Next, we subject the Pessimistic Cache Model to the same workloads and observe the reduction in the number of cache hits as expected [Figure 5-16]. However the reduction does not follow the pattern observed in the Optimistic Cache Model [Figure 5-12]. The effects of workload P3 and P4 (60% READs, and 40% READs correspondingly) is less for than the effect of the workload P5. It is also observed that the effect of P3, and P4 on the reduction in the number of cache hits is relatively equal for 70% of the workload, and then it diverges. Furthermore, observe that the workload P5, containing 80% WRITE requests achieves a number of cache hits approaching 0%.

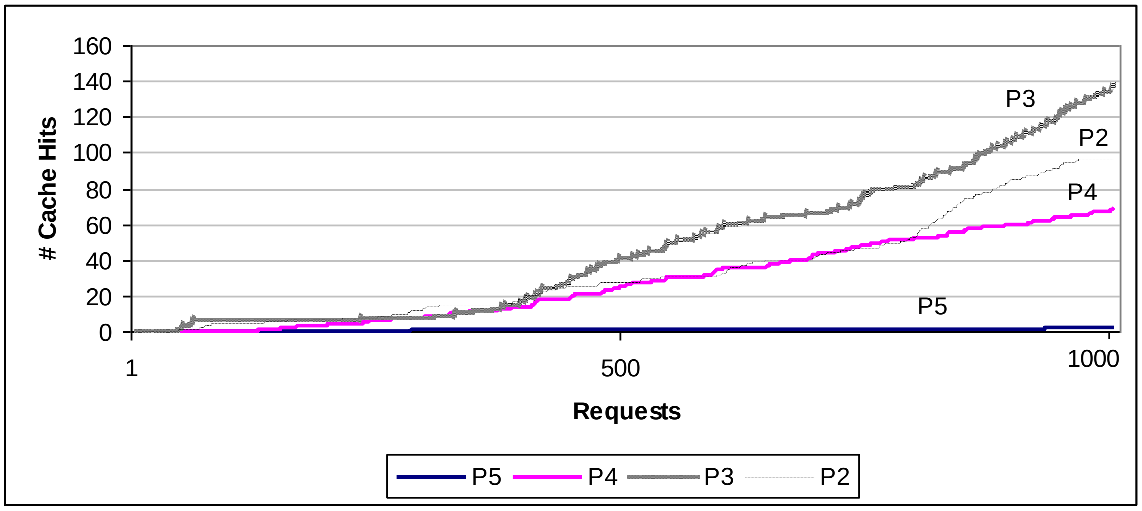


Figure 5-16. Effect of Workload on Cache Hits - Pessimistic Model

Next we observe the effect of the same workloads on the cache efficiency of the Pessimistic Cache Model [Figure 5-17]. The cache efficiency peaks later, proportionally to the lower number of READ requests in a workload. Observe the cache efficiency of P4 peaking after that of P3, which in turn peaks later that the peak of P2. The combination of the coarse granularity dependency matrix, and the workload P5 (containing only 20% READ operations) achieve almost a 0% efficiency. The result is attributed to the pessimistic behaviour of the model, resulting in a high rate of cache evictions, due to false positives as a result of the approximation of all resources by a singular R.

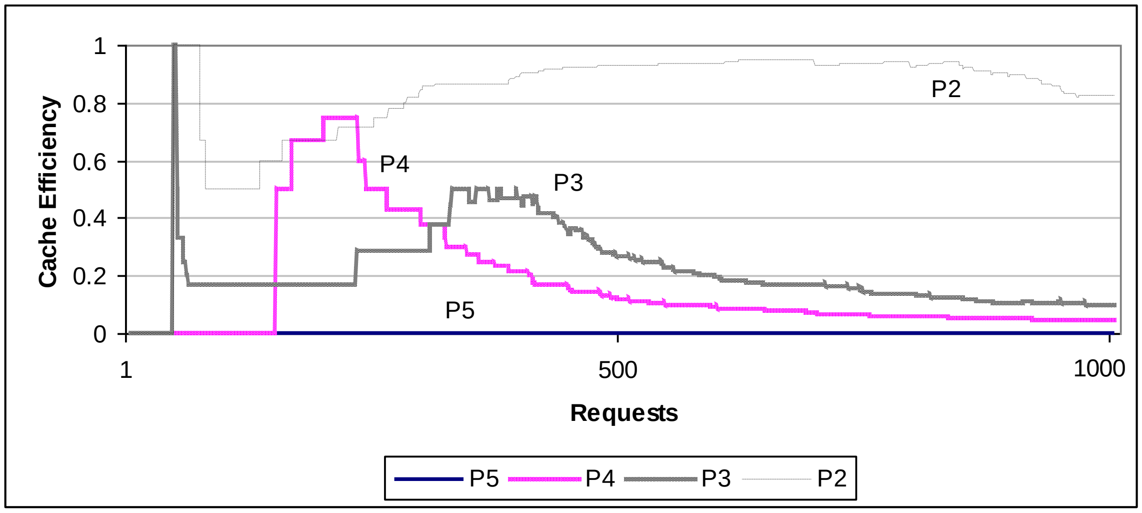


Figure 5-17. Effect of Workload on Cache Efficiency - Pessimistic Model

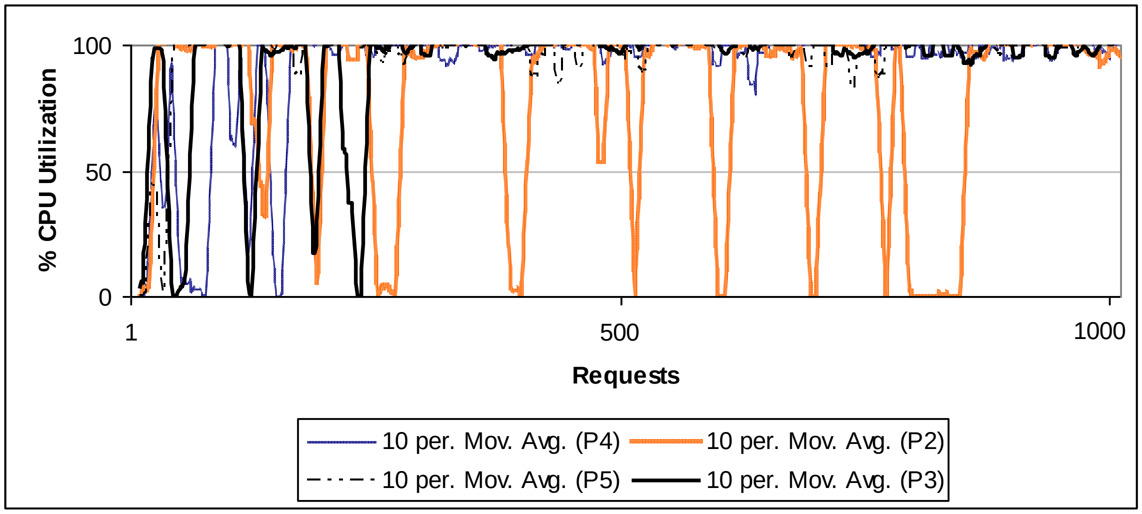


Figure 5-18. Effect of Workload on CPU Utilization on the client – Pessimistic Model

Next we observe the processing cost [Figure 5-18], represented by the percentage of CPU Utilization, to vary highly when using the workload P2. However the execution cost is dramatically increased by workloads containing higher numbers of WRITE operations. The processing cost of the dependency matrix, and pessimistic behavior is the primary factor in the observed result.

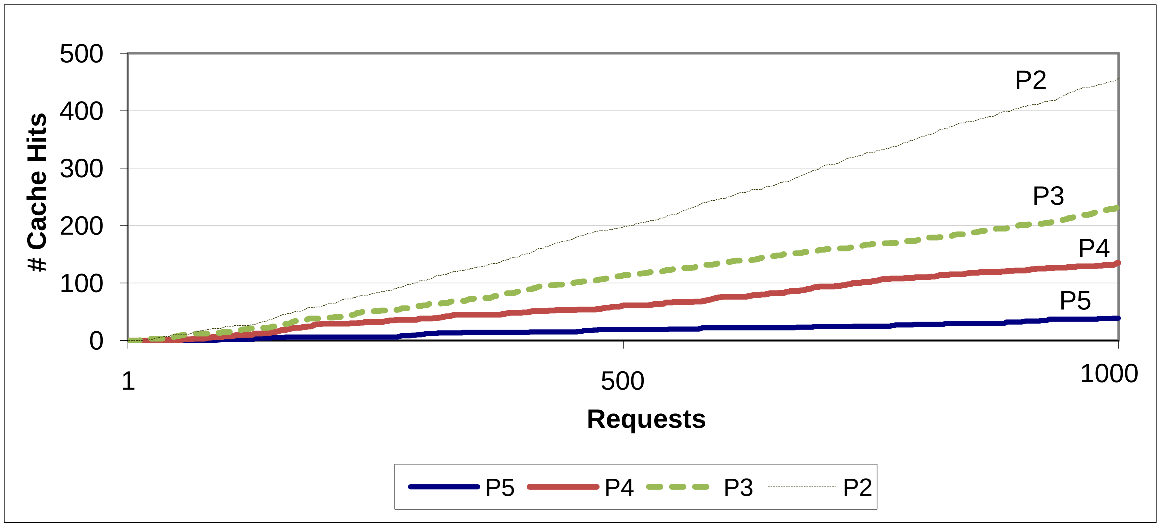


Figure 5-19. Effect of Workload on Cache Hits - Resource-Aware Model

In the following experiment we use the Resource-Aware cache model and observe the reduction in number of cache hits [Figure 5-19] when using workloads having decreasing numbers of READ requests. The reduction in cache hits is highest when transitioning from workload P2 to P3 (containing 80%, 60% READ requests, correspondingly).

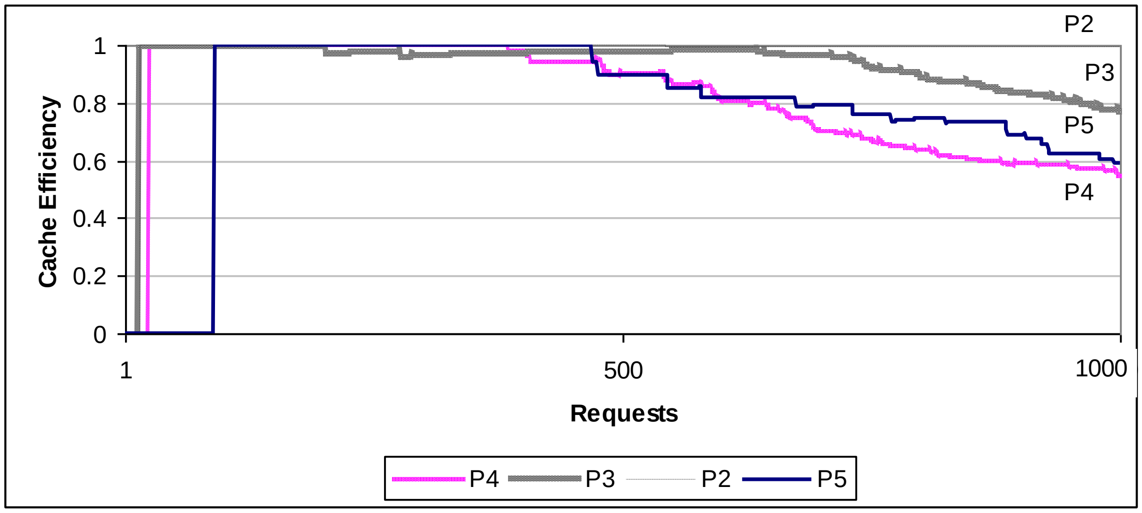


Figure 5-20. Effect of Workload on Cache Efficiency - Resource-Aware Model

The effect of workloads on cache efficiency [Figure 5-20] follows the pattern of later peaks observed when using the previous cache models. The peaks of cache efficiencies, however, continue until almost 30% of the workloads are executed. The Resource-Aware model, using all workloads, achieves cache efficiency approaching 100%, until after 30% of the workload execution. The observed efficiency is maintained near 100% for P2, and drops to 80% for P3, 60% for P4, and rises unexpectedly for P5. The workload P5 achieves a slightly better cache efficiency than P4, and at some point almost 10% better efficiency than that of the workload containing 20% more READ requests. However this result is insignificant because the number of cache hits achieved by P5 is less than half of the hits of P4 [see Figure 5-19].

Next we observe the effects of workloads on the processing cost of the Resource-Aware cache model [Figure 5-21]. The cost increases as the number of WRITE requests increase in the workloads. Observe the higher tendencies of processing the workloads P3, P4 and P5 vs. the processing cost of P2. The increase of processing costs is attributed to the complexity of resolving the fine granularity of the dependency matrix. The dependency matrix is used increasingly more often as the number of WRITE requests is increased.

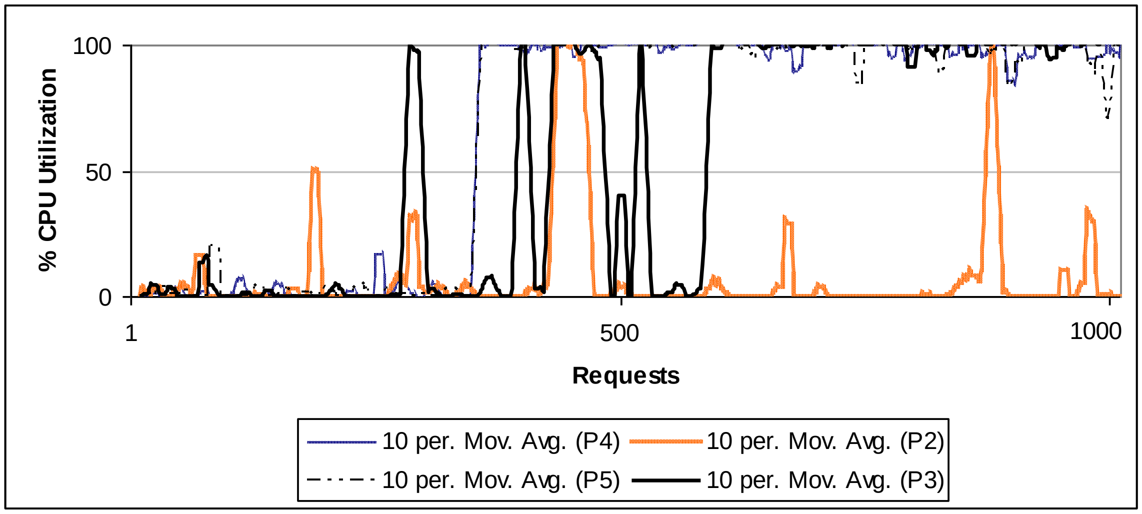


Figure 5-21. Effect of Workload on CPU Utilization on the client – Resource-Aware Model

Finally, we subject all cache models to connectivity patterns. The frequency of intermittent connectivity is increased by 20% for each connectivity pattern (e.g. C2 is 80% connected while C3 is 60% connected). Observe [see Figure 5-22] that the

Optimistic and the refined complexity cache models exhibit a degradation of cache efficiency as disconnectivity is increased. As the frequency of disconnectivity increases, we observe that cache efficiency decreases [see Figure 5-22]. The result is that shorter periods of connectivity, lead to slower consistency maintenance. Additionally, observe that cache hit rates decrease as disconnectivity is increased [Figure 5-23]. This observation is true for both the optimistic and the Resource-Aware cache models.

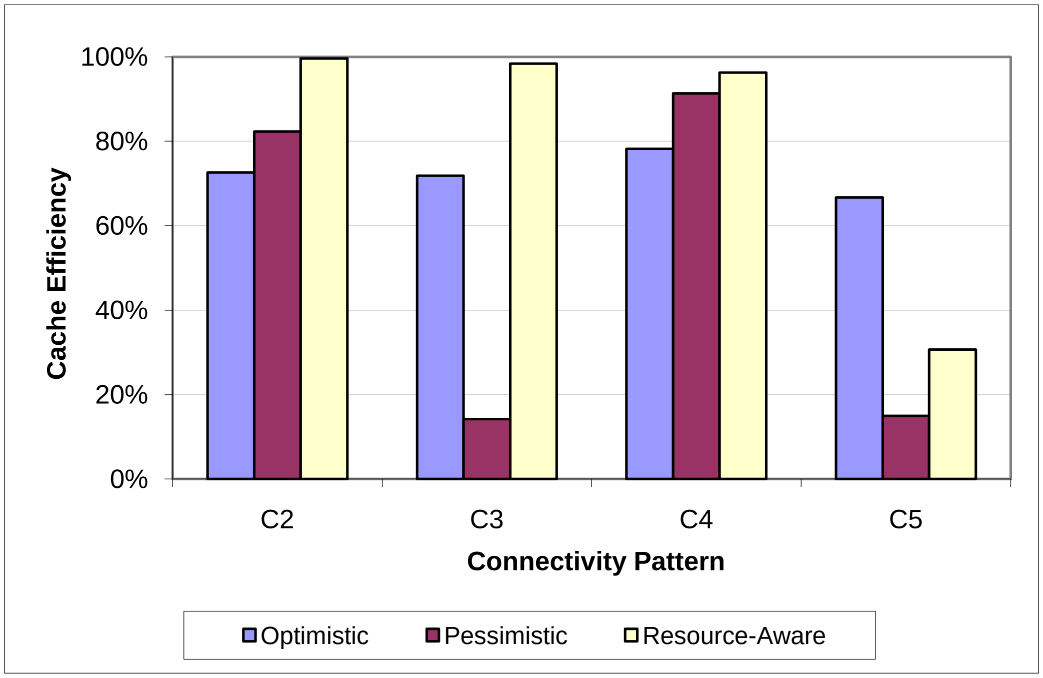


Figure 5-22. Effect of Connectivity Pattern on Cache Efficiency of cache models

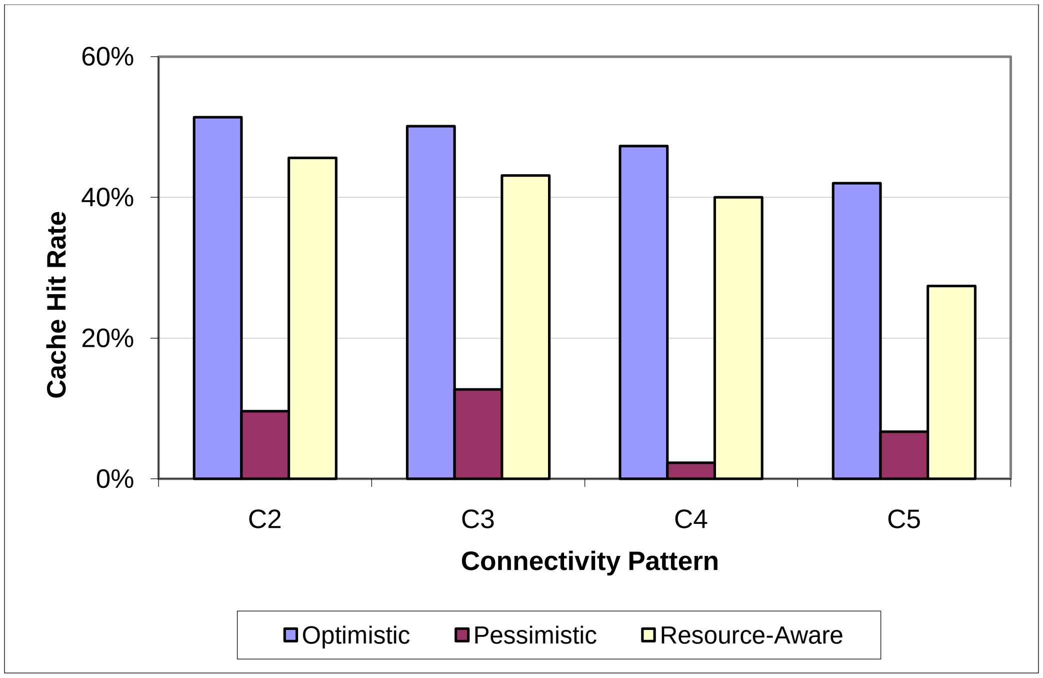


Figure 5-23. Effect of Connectivity Pattern on Cache Hit Rate achieved by cache models

The execution of the Pessimistic Cache Model produces mixed results (e.g. observe C3 vs C4). The cache efficiency of the Pessimistic Model varies significantly as the use of the coarse dependency matrix in the Pessimistic Cache Model leads to a large number of invalidated cached READs, on every new WRITE request. The pessimistic behaviour, in combination with the non-deterministic scheduling of Trickle Integration, significantly influences the consistency of the cache upon each experimental run. The results are magnified during long periods of disconnectivity (e.g. C4 and C5), as the results show [Figure 5-22, Figure 5-23].

CHAPTER 6
CONCLUSION

The prevalence of Service Oriented Architecture in the future of computing is highlighted by its support for client/service patterns in heterogeneous environments. In addition, the increasing availability and advances of mobile computing motivate supporting the nomadic use of WS clients.

This research presents a framework, and sample implementation (for Windows Mobile) enabling seamless operation of nomadic WS clients. The framework has low cost when using a simple model and leads to a reduction of request latency. The framework's design relies on semantics supporting the caching of WS interactions, while eliminating the cost and risks of SOAP processing. We've shown, using three cache models that the framework enables disconnected operation of the mobile WS client. The carefully chosen location of the framework decreases the costs of SOAP parsing/processing, avoids cache busting, and operates transparently of the client application's logic.

The specification format of Cache semantics, as XML, enables future extensions to support a tailored mobile cache. Two core semantics are supported by the Cache Semantics Description, the first is cacheability and the second is consistency maintenance. The cacheability semantic is the minimum requirement of document specification. The consistency maintenance semantic details the functional dependencies of WS methods and resources, specified at either coarse granularity (by approximation of resources), or specified at high granularity (by considering method arguments).

The results show that a simple cache model, where only the cacheability semantic is known, can be used to protect a WS client from intermittent connectivity at a low cost. However, because the cache model is optimistic on the validity of cache records (as semantics supporting consistency maintenance are not known), then the model is recommended for use only in short periods of intermittent connectivity. The results show that the Optimistic cache model is sufficient to reach 70% cache efficiency, during periods of infrequent disconnectivity.

A cache model that specifies the consistency maintenance semantic at coarse granularity is worse off than the simple cache model. The coarse granularity of consistency maintenance leads to pessimistic behavior where many READ records are invalidated by every WRITE, leading to a severe decrease in the number of cache records which in turn degrade the cache hit rate, and cache efficiency. The processing cost of the pessimistic model is also significantly higher than the processing cost of the simple optimistic model, because of the cost of consistency maintenance and the higher cost of reintegration.

The results show that to maintain cache efficiency during intermittent connectivity, the specification of cacheability must be associated with semantic supporting cache consistency maintenance. The Resource-aware cache model specifies finely grained consistency maintenance semantic, and it is shown to significantly improve cache efficiency above all other models. The cache efficiency of the Resource-aware model reaches 99%, at a good cache hit rate (comparable to the cache hit rate of the simple optimistic cache model).

The results also show that Cache Efficiency is reduced in extended periods of null connectivity. Additionally, when using the Pessimistic/Resource-Aware cache models: workloads containing a high percentage of WRITE requests have a higher processing cost.

The results showed the effect of using models of increasing complexity on cache efficiency. It is shown that as model complexity increases and more consistency maintenance is performed that the cache hit rate decreases. This result is explained by the smaller number of records remaining in the cache, as a result of consistency maintenance.

The negative effect of extended periods of disconnectivity on cache models was also shown. Long periods of intermittent connectivity reduce the amount of time available for consistency maintenance during reintegration. A direct result is that cache efficiency degrades over time. However, the effect of periods of disconnectivity, in combination with the scheduling of background reintegration is significant on the Pessimistic cache model. The pessimistic behavior of the model leads to longer queues, to be processed during reintegration, highlighting the need for longer periods of connectivity. Using the Pessimistic cache model highlights that Trickle Integration, combined with pessimistic behaviour, results in significant variation of results. As a result, the use of this model is not recommended.

Finally, the results show that workload characteristics (the ratio of READ to WRITE requests) have an effect proportional to the decrease of number of achievable cache hits. The lower the number of READs in a workload, the lower is the cache hits. Furthermore, we've shown that workloads containing large percentage of WRITE requests have an increasingly higher cost when using cache models that specify the dependency matrix element. The cost is a result of the expense of model processing, and in addition to cost of reintegration.

The use of the complex, Resource-aware cache model is expensive, because of the processing costs, however its use is less expensive than the use of the Pessimistic cache model. As advances in mobile computing reduce the cost effect (e.g. multi-core processing units), and because the achieved cache efficiency of the Resource-aware model is significantly better (reaching 99%), the use of this model is recommended.

The experiments and results are subject to the design of the framework, and to the unavailability of real user traces, and recorded connectivity patterns.

Future Work

1. Use additional synthetic workloads to determine the effect of randomness on experiments and conclusions.

2. Use of workloads composed of user traces, and connectivity patterns recorded by connectivity monitor to test the framework. The experiments lend support to the utility of the framework in a mass deployment setting.

3. Implement the framework on other platforms (e.g. Google Android, Apple iPhone).

4. The nomadic unit is subject to battery life limitations. A cold restart of the nomadic unit, during period of null connectivity, renders the cache ineffective. To overcome the effect of cold start, the framework can be assisted by knowledge of the remaining battery life. The cache content can be saved once an off period is predicted. Upon a recharge, the mobile cache resumes from its previous state, resulting in more cache hits due to a warm start.

5. The framework can benefit from monitoring the patterns of human movement, especially in regards to crowd movement. Disconnected cache hits could be improved if

the mobile cache can cooperate and share entries with nearby caches. Additionally, it may be possible to augment shared entries with additional information. The shared information may support inference of network quality of service, in relation to movement. This approach can assist the mobile client in predicting periods of null connectivity and to use localized QoS data to optimize its operation (e.g. perform early consistency maintenance, or to prefetch entries before a disconnection occurs).

6. The framework currently relies on the human specification of cache semantics, per web service. A mobile client will only benefit from the framework once the semantics are specified. Alternatively, if a local component can monitor the patterns of WS communication of local applications, then it maybe possible to automatically assign cacheability semantics to WS operations. Additionally, the special component monitors and cooperates with a central repository for automatic discovery of semantics descriptions. The goal is to enable mobile WS applications to function after minimum modification.

LIST OF REFERENCES

E. ALWAGAIT and S. GHANDEHARIZADEH. DeW: A Dependable Web Services Framework. International Workshop on Research Issues on Data Engineering: Web Services for E-Commerce and E-Government Applications, pp. 111-118, 2004. http://dblab.usc.edu/Users/shkim/papers/Dew-RIDE.pdf

APACHE SOFTWARE FOUNDATION, 2005: http://www.apache.org/

M. BAENTSCH, L. BAUM, G. MOLTER, S. ROTHKUGEL, and P. STURM. Enhancing The Web Infrastructure - From Caching To Replication. IEEE Internet Computing, vol. 1, no. 2, pp. 18-27, 1997. http://dx.doi.org/10.1109/4236.601083.

D. BARBARA and T. IMIELINSKI. Sleepers and Workaholics: Caching Strategies in Mobile Environments (Extended Version). VLDB Journal, vol. 4, no. 4, pp. 567-602, 1995. http://www.vldb.org/journal/VLDBJ4/P567.pdf

X. CAO and R. BUNT. An Architecture to Support Adaptive Mobile Applications. Proceedings of Wireless 2002, pp. Q-1, 2002. http://www.cs.usask.ca/ftp/pub/discus/recent_2002/cao-wireless2002.ps

P. CAO, E. FELTEN, A. KARLIN, and K. LI. A Study of Integrated Prefetching and Caching Strategies. ACM SIGMETRICS Performance Evaluation Review, vol. 23, no.1, pp. 188-197, May 1995. http://doi.acm.org/10.1145/223586.223608

P. CAO, J. ZHANG and K. BEACH. Active Cache: Caching Dynamic Contents on the Web. International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware), pp. 373-388, 1998. http://www.iop.org/EJ/abstract/0967-1846/6/1/305

CDMA DEVELOPMENT GROUP, 2005: http://www.cdg.org/

E. CERAMI. Web Services Essentials. O'Reilly & Associates, ISBN 0-596-00224-6, ch. 6, fig. 6-1, 2002.

K. CHEN, R. BUNT, and D. EAGER. Write Caching in Distributed File Systems. IEEE International Conference on Distributed Computing Systems, vol. 0, no. 0, pp. 457-466, 1995. http://citeseer.ist.psu.edu/chen95write.html

G. CHOCKLER, D. DOLEV, R. FRIEDMAN and R. VITENBERG. Implementing a Caching Service for Distributed CORBA Objects. In Proceedings of Middleware'00, pp. 1-23, 2000. http://citeseer.ist.psu.edu/chockler00implementing.html

J. CHUNG, K. LIN, and R. MATHIEU. Web Services Computing: Advancing Software Interoperability. ACM Computer, vol. 36, no. 10, pp. 35-37, October 2003. http://csdl2.computer.org/comp/mags/co/2003/10/rx035.pdf

D. CONAN, S. CHABRIDON, O. VILLIN, and G. BERNARD. Domint: A Platform for Weak Connectivity and Disconnected CORBA Objects on Hand-Held Devices. Technical Report, Institut National des Télécommunications, Évry, France, 2003. http://www-inf.int-evry.fr/~conan/Publications/2003_tech_report.pdf

K. DEVARAM and D. ANDRESEN. SOAP Optimization via Client-side Caching. Proceedings of the First International Conference on Web Services - ICWS'03, pp. 520-524, 2003. http://www.cis.ksu.edu/~dan/despot/icws03.pdf

W. EMMERICH. Engineering Distributed Objects. John Wiley & Sons, ISBN: 0471986577, 2000.

R. FRIEDMAN. Caching Web Services in Mobile Ad-Hoc Networks: Opportunities and Challenges. Proceedings of the second ACM international workshop on Principles of mobile computing, pp. 90-96, 2002. http://doi.acm.org/10.1145/584490.584508

K. FROESE and R. BUNT. Scheduling Write Backs for Weakly-Connected Mobile Clients. Proceeding of Computer Performance Evaluation: Modelling Techniques and Tools, 10th International Conference, vol. 1469, pp. 219-230, 1998. http://www.cs.usask.ca/ftp/pub/discus/old_2002/paper.98-1.ps.Z

C. HUANG, S. SEBASTINE, and T. ABDELZAHER, 1999. An Architecture for On-Demand Active Web Content Distribution. http://citeseer.ist.psu.edu/686616.html

IBM ECLIPSE, 2005: http://www.eclipse.org/

IEEE 802.11, The Working Group for Wireless LANs Standards, 2005: http://grouper.ieee.org/groups/802/11/

A. JOSEPH, J. TAUBER, and M. KAASHOEK. Mobile Computing with the Rover Toolkit. IEEE Transactions on Computers, vol. 36, no. 3, pp. 337-352, 1997. http://www.pdos.lcs.mit.edu/papers/toc.ps

J. KISTLER and M. SATYANARAYANAN. Disconnected Operation in the Coda File System. Proceedings of the thirteenth ACM symposium on Operating systems principles, pp. 213-225, ACM Press, 1992. http://doi.acm.org/10.1145/121132.121166

L. KLEINROCK. Nomadic Computing - An Opportunity. ACM SIGCOMM Computer Communication Review, vol. 25, no. 1, pp. 36-40, 1996. http://citeseer.ist.psu.edu/kleinrock95nomadic.htm

T. LA PORTA, K. SABNANI, and R. GITLIN. Challenges for Nomadic Computing: Mobility Management and Wireless Communications. ACM Journal of Nomadic Computing, vol. 1, no. 1, pp. 3-16, 1996. http://citeseer.ist.psu.edu/106356.html

C. LINDEMANN and O. WALDHORST. Consistency Mechanisms for a Distributed Lookup Service supporting Mobile Applications. Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, pp. 61-68, 2003. http://doi.acm.org/10.1145/940923.940936

A. MAHANTI, D. EAGER, and C. WILLIAMSON. Temporal locality and its impact on Web proxy cache performance. Special issue on internet performance modeling, vol. 42 , no. 2-3, pp. 187-203, 2000. http://portal.acm.org/citation.cfm?id=360757

MICROSOFT VISUAL STUDIO, 2005: http://msdn.microsoft.com/vstudio/

E. NEWCOMER and G. LOMOW. Understanding SOA with Web Services. Addison-Wesley, ISBN 0321180860, 2004.

B. NOBLE, M. SATYANARAYANAN, D. NARAYANAN, J. TILTON, J. FLINN, and K. WALKER. Agile Application-Aware Adaptation for Mobility. Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint-Malo, France, 1997. http://citeseer.ist.psu.edu/noble97agile.html

OASIS, Reference Model for Service Oriented Architectures, 2005: http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm

M. 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

M. SATYANARAYANAN. Mobile Information Access: Accessing information on demand at any location. IEEE Personal Communications, vol. 3, no. 1, 1996. http://citeseer.ist.psu.edu/article/satyanarayanan96mobile.html

W. SCHULTE and Y. NATIS. Service Oriented Architectures, Part 1, Research Note SPA-401-068, 1996. http://www.gartner.com/DisplayDocument?doc_cd=29201&ref=g_fromdoc

T. TAKASE and M. TATSUBORI. Efficient Web Services Response Caching by Selecting Optimal Data Representation. Proceedings of ICDCS, vol. 3, pp. 24-03, 2004.

T. TAKASE, Y. NAKAMURA, R. NEYAMA and H. ETO. A Web Services Cache Architecture Based on XML Canonicalization. Poster on Eleventh International World Wide Web Conference, 2002. http://www2002.org/CDROM/poster/126/

D. TERRY and V. RAMASUBRAMANIAN. Caching XML Web Services. ACM Queue, vol. 1, no. 3, pp. 70-78, March 2003. http://doi.acm.org/10.1145/846057.864024

M. TIAN, T. VOIGT, T. NAUMOWICZ, H. RITTER, and J. SCHILLER. Performance Considerations for Mobile Web Services. Workshop on Applications and Services in Wireless Networks, Bern, Switzerland, 2003. http://www.sics.se/~thiemo/aswn2003.pdf

W3C, Hypertext Transfer Protocol (HTTP/1.1), RFC 2616, 2005: http://www.w3.org/Protocols/rfc2616/rfc2616.html

W3C, Simple Object Access Protocol (SOAP 1.1), 2005:
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

W3C, WEB SERVICES ACTIVITY, 2005: http://www.w3.org/2002/ws/

Z. WANG, S. DAS, H. CHE and M. KUMAR. Scalable Asynchronous Cache Consistency Scheme (SACCS) for Mobile Environments. IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 11, 2004.
http://crystal.uta.edu/~hche/PUBLICATIONS/papers/TPDS-0048-0403-1.pdf

J. WANG. A Survey of Web Caching Schemes for the Internet. ACM Computer Communication Review, vol. 25, no. 9, pp. 36-46, 1999.
http://citeseer.ist.psu.edu/wang99survey.html

S. WEERAWARANA, F. CURBERA, F. LEYMANN, T. STOREY and D. FERGUSON. Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More. Prentice Hall PTR, ISBN 0131488740, 2005.

C. WILLIAMSON. Wireless Web Performance Issues. Book chapter, 2005.
http://pages.cpsc.ucalgary.ca/~carey/papers/2005/WirelessWeb.pdf