WSDAF: A Dynamically Adaptive Framework for Scalable Web Services

Congchun He
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University
congchun@cs.nyu.edu

Advisor: Vijay Karamcheti

March 16, 2004
Abstract

The growing popularity of XML and Web Services is transforming the World Wide Web from its information centric roots into a programmable network offering a wide variety of computation and data services to its users. A growing number of applications are being constructed by assembling preexisting services, and typically accessed over a large scale, heterogeneous wide area network by users who are geographically distributed, using a variety of devices that present different quality of service requirements. To support such applications, there arise two important issues: ensuring good performance across wide area environments and the ability to do so while adapting to changes of usage patterns as well as network conditions.

This proposal describes a novel approach to address the above two issues. Our approach is to make use of a collection of predetermined locations which are distributed across the wide area network. The key functionality of these locations is to detect locality among the service requests over space† and time. This information is used to efficiently find an optimal and dynamically adjustable strategy for service replication. We present an implementation of our approach, WSDAF, a dynamically adaptive framework for scalable Web Services, which we use as a prototype. WSDAF leverages SOAP Routers to relay user requests, with embedded monitoring functionality to detect network changes and study usage patterns using a statistical model and a dynamically adaptive data structure. Our preliminary results are promising: WSDAF can efficiently detect the locality in workload, track the change of network conditions, and adapt to these changes with a significant improvement in performance.

†We use the term, “space”, to refer to the portions of network where users reside as well as the data or computational space represented by the services.
# Contents

1 Introduction ........................................... 1

2 Background and Related Work .......................... 2
  2.1 XML, Web Services and Related Technologies .............. 2
  2.2 Related Work I: Component-based Service Integration .......... 5
  2.3 Related Work II: Web Caching and Replication ............... 6
    2.3.1 Web Caching and Replication Architectures ............ 7
    2.3.2 Peer-To-Peer Caching .................................. 9
    2.3.3 Web Caching for Dynamic Content ..................... 9

3 Motivating Example ..................................... 12
  3.1 MapPoint Web Services .................................. 12
  3.2 Proposed Approach ..................................... 14

4 Architecture ........................................... 15
  4.1 Preliminaries ......................................... 15
  4.2 WSDAF Architecture .................................... 16
    4.2.1 Design Overview ...................................... 17
    4.2.2 WebService Registry ................................. 18
    4.2.3 SoapRequest Translator ............................... 18
    4.2.4 Access Monitor ...................................... 19
    4.2.5 Performance Monitor ................................. 20
    4.2.6 Caching Managers .................................... 20

5 Algorithms, Protocols and Policies ..................... 21
  5.1 Algorithms ........................................... 22
    5.1.1 Cell Splitting: Hit-count Adjustment Algorithm ........ 22
    5.1.2 Cache Placement Problem and Algorithms ................ 22
  5.2 Protocols and Policies .................................. 29
    5.2.1 Performance Monitoring Protocol ..................... 29
    5.2.2 Cache Replacement Policy ............................ 30

6 Implementation .......................................... 31

7 Performance Evaluation .................................. 31
  7.1 Experiment Setup ..................................... 32
    7.1.1 Experiment Environment .............................. 32
    7.1.2 Evaluation Methodology .............................. 32
  7.2 Performance Under Load ................................ 33
    7.2.1 Synchronized Workload .............................. 34
    7.2.2 Staggered Workload ................................. 37
    7.2.3 Performance of Origin Server ....................... 42
    7.2.4 Performance impact of caching policies ............... 43

8 Discussion ............................................. 45

Appendices ................................................ 47
# List of Figures

1. Motivating Example: MapPoint Web Services ............................................. 13
2. 2ⁿ-ary CellTree: The Representation of an n-dimensional Data Space .............. 16
3. An overview of the WSDAF Architecture .................................................. 16
4. Soap Request Processing on a WSDAF Router ........................................... 17
5. An example of Cell Splitting in 2-d Space ............................................... 20
6. A simple example of Cache Placement ...................................................... 24
7. Chain-based WSDAF Router Architecture ............................................... 25
8. Performance Monitoring Protocol .............................................................. 30
9. Network Setting for WSDAF Experiments ................................................. 32
10. Resp. times seen on an unloaded network (w/o caches) for 3 clients: C₀, C₁₆, and C₂₈ 34
11. Performance under Synchronized Workload with large α, large β .................. 35
12. Performance under Synchronized Workload with small α, large β ................. 36
13. Performance under Synchronized Workload with large α, small β ................. 38
14. Performance under Synchronized Workload with small α, small β ................. 39
15. Performance under Staggered Workload with large α, large β ..................... 40
16. Performance under Staggered Workload with small α, large β ..................... 40
17. Performance under Staggered Workload with large α, small β ..................... 41
18. Performance under Staggered Workload with small α, small β ..................... 42
19. Performance of WSDAF Router R₈ ............................................................ 42
20. Performance impact of request repression policy ........................................ 43
21. Performance impact of cache replacement policy adjustment ...................... 44
22. Performance impact of cache replication policy adjustment ........................ 44
1 Introduction

The growing popularity of XML and Web Services is transforming the World Wide Web from its information centric roots into a programmable network offering a wide variety of computation and data services to its users. As fundamental building blocks, Web Services together with standard, widely accepted protocols such as SOAP, UDDI and WSDL, provide an infrastructure for distributed computing where sophisticated applications can be constructed by integrating various preexisting services. As illustrative examples, one may think about commercial service providers like Amazon [1] and Google [2], and online imagery providers like MapPoint [3], TerraServer [4] and SkyServer [5], etc.

Applications such as the ones above are typically accessed over a large scale, heterogeneous wide area network by users who are geographically distributed, using a variety of devices that present different quality of service requirements. Hence, there arise two important issues: ensuring good performance across wide area environments and the ability to do so while adapting to changes of usage patterns as well as network conditions. Web performance is especially crucial for e-commerce environments: a Zona Research [6] study in 2001 revealed that with response times of less than 7 seconds, e-commerce sites find 7% abandonment rates, whereas this rate increases sharply to 70% with response times greater than 12 seconds. The long response delay and the consequent high abandonment rates lead to severe revenue losses in e-commerce environments: in 1998, it was estimated to be 1.9 billion dollars [7].

A common solution to the first issue is to move services as close to end-users as possible. However, in heterogenous wide area networks, service offloading is not free and the cost could increase as the distance between the replica and the origin increases. Therefore, one needs a distributed service replication algorithm to determine what to replicate and where to place the replicas with minimum cost. The second issue comes from the observation that network conditions may change over time, and usage patterns may vary in the nature of requests over space and time. Since these changes usually happen in a small time scale, the infrastructure should be adaptive in an automatic fashion.

This proposal describes a novel approach to address the above two issues. Our approach is to make use of a set of predetermined locations which are distributed across the wide area network. Each location acts as an intermediary in relaying requests to Web Services, usually SOAP messages, and is able to monitor the distributed requests from users in diverse geographical areas as well as any change in network conditions. The key functionality of these locations is to detect locality among the service requests over space and time. The locality is used to facilitate modelling service usage patterns; identifying portions of the service that are highly accessed over space and time. There are two kinds of locality: spatial and temporal. An example in reality for the former is the MapPoint Web Service where users in a specific geographical area are apt to request maps around that area, representing a spatial locality. For the latter, one may think about news services where whenever a breaking news happens, thousands or millions of requests for the same objects might occur in a small time scale, representing a temporal locality. Such locality information can be used to efficiently find an optimal strategy for service replication. However, before we can ensure that this approach is effective, the following questions need to be addressed: (1) Does locality exist in the underlying workload distributed across the network? (2) Is there an efficient mechanism to dynamically detect such locality? (2) How to design a distributed algorithm to solve the service offloading problem? (3) How to design communication protocols for distributed locations to exchange information needed by the distributed algorithm?

We present an implementation of our approach, WSDAF, a dynamically adaptive framework for scalable Web Services, which we use as a prototype. WSDAF leverages SOAP Routers to
relay user requests, with embedded monitoring functionality to detect network changes and study usage patterns using a statistical model and a dynamically adaptive data structure. WSDAF has also developed communication protocols to exchange information between SOAP Routers and a distributed algorithm to solve the service replication problem. We evaluate the performance of WSDAF by applying it onto a simulated system modelling the MapPoint Web Service. Our preliminary results are promising: WSDAF can efficiently detect the locality in workload, track the change of network conditions, and adapt to these changes with a significant improvement in performance.

The rest of the proposal is organized as follows. In Section 2, we give an overview of the background and related work. In Section 3, we discuss a motivating example for our work. In Section 4, we describe the architecture of WSDAF and introduce the core components used in the infrastructure. In Section 5, we present the algorithms for request monitoring and service replica deployment, the protocols defined for communication between WSDAF components, and the policies used for QoS assurance purposes. The implementation details of WSDAF are given in Section 6 and the performance evaluation of our infrastructure on a simulated system is in Section 7. Finally, we conclude this proposal and discuss a plan for future work in Section 8.

2 Background and Related Work

2.1 XML, Web Services and Related Technologies

Part I

XML [11], standing for eXtensible Markup Language, describes a class of data objects called XML documents and partially describes the behavior of computer programs which process them. As a markup language that allows users to define their own customized tags, XML is platform-independent, language-independent and media-independent. Therefore it is easier for systems in different environment to exchange information using XML. The universality of XML makes it a very attractive way to communicate information between data-centric Web applications, such as database exchange, distribution of processing and business transactions. Furthermore, XML together with XML Namespaces [12] and XML Schemas [13], provides useful mechanisms to deal with structured extensibility in a distributed environment. XML Namespaces provide a simple method for qualifying element and attribute names used in XML documents by associating them with namespaces identified by IRI references, where “an IRI reference is a string that can be converted to a Uniform Resource Identifiers(URI) reference by applying a set of rules” [12]. XML Schemas express shared vocabularies and allow machines to carry out rules made by people by providing a means for defining the structure, content and semantics of XML documents.

A Web Service [18] is programmable application logic accessible using standard Internet protocols. Similar to components, Web Services provide black-box functionality that can be reused without knowledge of the implementation details. The difference between Web Services and current component technologies is that Web Services are not accessed via object-model-specific protocols, such as DCOM, RMI, or IIOP. Instead, they are accessed via standard, widely accepted web protocols and data formats. In most cases, the web protocol is HTTP and the data format is XML. Web Services that use XML as their data format are referred to as XML Web Services.

XML Web Services are becoming the fundamental building blocks in the move towards distributed computing in the World Wide Web. By using open standards and focusing on communication and collaboration more than on service internals, XML Web Services have become a popular
platform for application integration: sophisticated Web applications can be constructed from a variety of preexisting XML Web Services without worrying about where they reside or how they were implemented. Although there may be a lot of definitions of XML Web Services, almost all definitions have the following properties in common:

- XML Web Services expose useful functionality to Web users through a standard Web protocol, in most cases, the protocol used is SOAP [14]

- XML Web services provide a way to describe their interfaces in enough detail to allow a user to build a client application to talk to them. This description is usually provided in an XML document called a Web Services Description Language (WSDL) document [23]

- XML Web services are registered so that potential users can find them easily. This is typically done using a protocol called Universal Discovery Description and Integration (UDDI) [24]

SOAP stands for Simple Object Access Protocol. The advent of SOAP is motivated by the fact that application communication using Remote Procedure Calls (RPC) for current component-based technologies has compatibility and security problems: firewalls and proxy servers will normally block this kind of traffic. A better way to communicate between applications is over HTTP, since HTTP is widely supported by all Internet browsers and servers and not blocked by firewalls.

As stated in the specification [14], “SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics.” SOAP defines a message structure consisting of an optional header and a mandatory body. The SOAP header provides a flexible mechanism for extending a message in a decentralized and modular way. Typical examples of extensions that can be implemented as header entries are authentication, transaction management, etc. SOAP is mainly used for passing documents: Electronic Document Interchange (EDI) and providing an XML-RPC-like request/response protocol.

WSDL, standing for Web Services Description Language, is an XML-based language used to define Web Services and describe how to access them. With WSDL, Web Services can be described as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. In most cases, the network protocol is HTTP and the message format is SOAP.

A WSDL file is essentially a XML document, therefore it is readable and editable by a human being. But in most cases, it is generated and consumed by software. There are several tools available to read a WSDL file and generate the code required to communicate with an XML Web Service. The tools can also generate WSDL files from existing programming interfaces.

UDDI, standing for Universal Discovery Description and Integration [24], is an industry specification for publishing and locating information about Web services. It defines an information framework that enables users to describe and classify their organizations, the services offered, and the technical details about the interfaces of the Web Services exposed. UDDI also defines a set of Application Programming Interfaces (APIs) that can be used by applications and services to interact with UDDI data directly. A UDDI directory entry is an XML file that describes a business and the services it offers. There are three parts to an entry in UDDI directory: “white pages”
describe the company information; “yellow pages” include industrial categories based on standard
taxonomies; “green pages” describe the interface to the service exposed. UDDI uses a document,
called Type Model or tModel, to define services. In most cases, the tModel contains a WSDL file
that describe a SOAP interface to an XML Web Service, although it is flexible enough to describe
almost any other service.

XSL and XSLT. XSL stands for eXtensible Style Sheet Language. In order to display XML
documents, it is necessary to have a mechanism to describe how the document should be displayed.
XSL is the preferred style sheet language of XML, performing the same role as Cascading Style
Sheets (CSS) in HTML. But XSL is far more sophisticated than CSS because XSL consists of two
The former, called XSL Transformations (XSLT), is the part of XSL that is used to transform an
XML document into another XML document, or another type of document that is recognized by a
browser, like HTML and XHTML. XSLT can also add new elements into the output file, or remove
elements. It can rearrange and sort elements, and test and make decisions about which elements
to display, and a lot more. A common way to describe the transformation process is to say that
XSLT transforms an XML source tree into an XML result tree.

Part II
So far, we have introduced XML, Web Services and the basic standards related to the two. As
we have seen, SOAP is the most common data format used by these technologies to exchange
information. However, the SOAP standard is incomplete with respect to two issues.

First, SOAP by itself does not define an actual message path along which a SOAP message is
to travel. In order to provide the semantics for actually exchanging messages, SOAP can be bound
to an application layer protocol such as HTTP or SMTP. These protocols define their own message
path models and message exchange patterns that in general differ from the SOAP message model.

Second, SOAP by itself does not define a mechanism for building secure Web Services with
implementation of integrity and confidentiality. In order to support secure message exchanging,
SOAP has to be used in conjunction with a variety of secure protocols and encryption technologies.

To address these incompatibility and inconvenience problems, several new advanced protocols
such as WS-Routing, WS-Referral and WS-Security, have been proposed and are becoming widely
accepted.

WS-Routing. stands for Web Services Routing Protocol. According to the specification [20],
WS-Routing is a “SOAP-based, stateless protocol for exchanging one-way SOAP messages from
an initial sender to the ultimate receiver, potentially via a set of intermediaries. In addition, WS-
Routing provides an optional reverse message path enabling two-way message exchange patterns
like request/response, peer-to-peer conversations, and the return of message acknowledgements and
faults.”

WS-Routing defines a mechanism that makes use of two types of message paths to exchange
messages: forward message path and optional reverse message path, where the former describes
how messages travel from the initial sender through zero or more intermediaries to the ultimate
receiver and the latter describes how messages travel in the direction from the ultimate receiver
through zero or more intermediaries to the initial sender. However, WS-Routing does not define
dynamic paths necessary for adaptive, real-time Web services traffic management.

WS-Routing relies on routing tables to determine the path that the message should take. An-
other protocol, WS-Referral, is used to update those routing tables.
**WS-Referral**, stands for **Web Services Referral** Protocol. As stated by the specification [21], WS-Referral is a “SOAP-based, stateless protocol for inserting, deleting, and querying routing entries in a SOAP router. A SOAP Router is a SOAP node that exposes SOAP message relaying as a Web service, either as a standalone service or in combination with other services”. An example of a SOAP router is a SOAP node supporting the WS-Routing Protocol.

In addition to mechanisms for inserting, deleting, and querying routing entries, WS-Referral defines an XML-based structure called a WS-Referral statement for describing a routing entry along with a set of conditions under which the statement is satisfied. Usually, the routing tables used by WS-Routing are provided as a configuration file containing WS-Referral statements. This configuration file is loaded by SOAP Routers at start-up time.

**WS-Security**, stands for **Web Services Security**. According to the specification [22], “WS-Security describes enhancements to SOAP messaging to provide quality of protection through message integrity, message confidentiality, and single message authentication. These mechanisms can be used to accommodate a wide variety of security models and encryption technologies.”

WS-Security also provides a general-purpose mechanism for associating security tokens (such as Kerberos tickets or X.509 certificates) with messages. No specific type of security token is required by WS-Security: it is designed to be extensible.

To use WS-Security, code modification on both client side and server side are required, which has prevented its wider acceptance.

### 2.2 Related Work I: Component-based Service Integration

Increasingly, scalable Web applications are being constructed by integrating service components that are individually developed. The web services architecture is one example of such a trend. The scalability and efficiency for such applications can be achieved by choosing and deploying these components appropriately across the network.

The proposed approaches in this trend include DCE [28], DCOM [29] and CORBA [31, 31], which rely on static component linkages, as well as CANS [33], Ninja [30], OGSA [35], and PSF [36] which aim at advocating a dynamic model for component integration at run-time. In this section, we discuss two representative approaches: OGSA and PSF.

**Open Grid Services Architecture**

The Open Grid Services Architecture (OGSA) [35] explores the advantages of integrating Grid technologies and Web services. This architecture defines semantics of Grid services and related mechanisms for creating, naming, and discovering transient Grid service instances; provides location transparency and multiple protocol bindings for service instances; and supports integration with underlying native platform facilities. OGSA also defines, in terms of WSDL interfaces and associated conventions, mechanisms required for creating and composing sophisticated distributed systems, including lifetime management, change management, and notification. Service bindings can support reliable invocation, authentication, authorization, and delegation, if required.

**Partitionable Services Framework**

The Partitionable Services Framework (PSF) [36] addresses the absence of adaptivity to the performance and security characteristics of the heterogeneous environment for service integration in
OGSA. PSF proposes a novel approach which enables services to be flexibly assembled from multiple components. The approach also facilitates transparent component migration and replication on demand at locations closer to end-users. The framework relies on four key elements: (1) a declarative specification of the service components; (2) a monitoring module; (3) a planning module, which steers the deployment to accommodate underlying environment characteristics; and (4) a deployment infrastructure which realizes the service deployment.

PSF integrates a decentralized trust management and access control system called dRBAC [34] with a programming and run-time abstraction called views [37], to provide a unifying mechanism for cross-domain authentication and authorization and support of single sign-on and fine-grained access control.

Although the infrastructures of component-based service integration discussed above can gracefully handle service naming, discovery and dynamical assembly, as well as resource allocation, authentication and authorization, and lifetime management, they do not address the two issues pointed out in Section 1. Components are usually viewed as black boxes, hiding internal computation and information about usage of data from outside. So it is difficult for the infrastructure to retrieve information about component usage patterns, which is needed in order to reason about cost of interaction between components as well as cost of component replication, if applicable. Furthermore, component deployment in such systems has typically been determined in a centralized fashion. In addition to the scaling challenges this presents, centralized schemes may also make it difficult to detect (and therefore exploit) locality among service requests. Consider for instance a MapPoint-like service, whose users in different states of U.S. are apt to request maps around their states, presenting spatial locality in database usage on state level, however, the central database might have a view that the requests are uniformly distributed among the states in America and fail to capture the existence of such locality.

2.3 Related Work II: Web Caching and Replication

The other main research trend aims at providing a scalable, high-performance Web Services infrastructure by means of service offloading. Generally speaking, web applications are designed to provide services involving both computation and data processing and delivery. Hence, any approach using service offloading needs to take both into consideration. The proposed approaches in this aspect include approaches for static content delivery such as [38, 41, 39, 42, 43, 46], and approaches for dynamic content caching such as [68, 73, 74, 70, 76].

Web content delivery has attracted a lot of interest in both industrial and academic communities. A simple way to improve content delivery performance is to upgrade the loaded resource: a faster server, a bigger switch, reengineering the network. Unfortunately, such an approach is not always economically feasible and more importantly, it does not consider that today a single web service may consist of many other services that reside in different locations. A better solution is to move web contents closer to end users to reduce latency and improve utilization of network bandwidth. These approaches include caching and replication.

Caching has traditionally been applied in distributed file systems such as AFS [51]. A cache is a temporary storage location containing a set of most commonly accessed objects copied from original servers. A Web cache is a dedicated system which monitors the page or object requests and stores the retrieved pages or objects from the server. On subsequent requests, the cache can serve them from its storage rather than passing requests to the origin server. Studies show that people are apt to access the same popular pages or objects from billions of pages or objects on the web. The advantages of moving such popular objects closer to users are obvious: network latency
is reduced, the fixed bandwidth of the upstream link is better utilized, server load is alleviated, etc. Although caching has been a proven technique for improving scalability and performance in distributed file systems, its application to the World Wide Web requires solutions to new problems, for example, the deployment of caches, the consistency of cached data and how to handle dynamic contents, etc.

Replication is a technique similar to caching. Replication has been commonly applied to distributed systems to improve availability and fault tolerance. Unlike caching, which attempts to store, in a demand-driven fashion, the most commonly accessed objects as close as possible to end-users, presenting a small granularity of data space, replication distributes, in a more or less static fashion, a site’s contents across multiple mirror servers, presenting a large granularity of data space. Relying on an efficient algorithm for load balancing, replication systems have very high fault tolerance since in the case that one replica crashes, other replicas or the original server can take over its role to serve requests. Compared with caching, replication is considered to be a mechanism with low flexibility, poor adaptability, increased space consumption and increased difficulty in maintenance. Some recent work such as “Materialized Views” tries to reduce the amount of replicated contents by defining a “view”, which represents a portion of origin database and replicating only that particular portion. However, this mechanism needs to anticipate load or work patterns in order to define appropriate “views”, and requires a lot of manual interference in the case that load or work patterns change. [84] presents an early overview on replication and its challenges.

As discussed above, the main distinction between caching and replication has to do with the mechanism (static or dynamic) and the granularity of distribution (small or large). A real application typically involves elements of both approaches, so in the rest of this proposal, we use these terms interchangeably.

2.3.1 Web Caching and Replication Architectures

In this section, we discuss several distinct web caching and replication architectures.

Proxy Caching

A proxy caching server intercepts requests from clients and either generates responses if the requests hit the cache or forwards them to the origin server on behalf of the users, retrieves the responses from origin server, possibly stores them into the cache, and finally returns to the clients. A proxy server is usually deployed at the edge of a network, called edge server, in order to serve a large number of users internal to an organization, and results in increased availability of static web contents, significant reduction of network latency, and wide area network bandwidth savings. One disadvantage to this design is that the proxy server represents a single point of failure in the network in that if the proxy crashes, the wider web is unavailable to all of the internal users. A proxy caching approach also requires that clients be manually configured to use the appropriate proxy server.

There are two major varieties of proxy caching, namely Reverse Proxy Caching and Transparent Caching. Reverse Proxy Caching deploys caches near the origin of contents, as opposed to the client side, such that even the server expects a large number of requests, it can still provide a high level of quality of service. With Transparent Caching architecture, users are not required to explicitly configure their web browsers. Transparent Caching relies on a HTTP filter which redirects outgoing HTTP requests to web cache servers or cache clusters.

A commercial approach to Proxy Caching, Akamai [49], provides a more general solution. This approach does not require the client to be configured to use a specific caching or proxy server. Instead, the content on the origin site is rewritten such that the embedded links point to a nearby
Akamai server. The client retrieves the main page from the origin site and follows the embedded links to retrieve embedded objects from Akamai servers. Additionally, by using DNS redirection mechanism, the Akamai infrastructure can control the specific server that is selected. By deploying over 14,000 servers in 1,100 networks in 70 countries and storing the site content of popular providers such as YAHOO, Akamai is able to reduce network latency in a dramatic way.

Hierarchical and Distributed Caching

A single cache has a finite size, and therefore there is a limit to the number of contents that can be cached. A group of caches where every cache shares its cached contents with others can in principle provide better performance. Such caches reside across the network and are organized according to some topology, which can be grouped into two broad categories: Hierarchical Caching and Distributed Caching.

In Hierarchical Caching scheme, caches are placed at multiple levels of the network, establishing neighborhood relationships with other caches where a parent cache is essentially one level up in a cache hierarchy. A client’s request is forwarded up the hierarchy until a cache hit occurs, or, if none occurs, the request is forwarded to the origin server. In [53, 83], several disadvantages of Hierarchical Caching scheme are identified: (1) The multiple levels of hierarchy always introduce additional overhead; (2) The high level caches in hierarchy may become bottlenecks; and (3) The same content being stored at different cache levels introduces redundant storage.

Hierarchical Web Caching was first proposed in the Harvest project [52]. Other examples include Adaptive Web Caching [55], Access Driven Web Caching [56], etc. Adaptive Web Caching targets the “Hot Spot” problem where a web server suffers a sudden surge of traffic due to some particular content being massively accessed. Adaptive caching consists of multiple, distributed caches which dynamically join and leave cache groups based on the content demand. Adaptive Web Caching developed the Cache Group Management Protocol (CGMP) and the Content Routing Protocol (CRP) to provide a caching scheme that is adaptable and self-organizing for heterogeneous demand of web contents.

Distributed Caching was designed as a complementary approach and allows the distribution of caching proxies geographically over large distance. In Distributed Caching, there no longer exist intermediate cache levels: each cache server resides at the bottom level in the network and maintains meta-data information about the cached contents on other caches, used to find the cache that contains the requested contents. For the purpose of distributing the meta-data information efficiently and scalably, a hierarchical distribution mechanism can be applied. Since most requests can be served in the lowest level of the network, Distributed Caching can reduce network latency significantly. The elimination of intermediate cache levels alleviates the redundant storage problem; the distribution of the meta-data information allows better load sharing; and together, they provide a more fault tolerant scheme. The main disadvantages in exploiting Distributed Caching include: (1) high penalty for a “cache miss” due to high connection time between cache servers; (2) the need for additional network bandwidth to distribute meta-data information; and (3) the administration for such a system is complicated.

There are several approaches for the Distributed Caching scheme, including the Internet Cache Protocol (ICP) [57] in Harvest project, the Cache Array Routing Protocol (CARP) [58], the distributed Internet Cache by Provey and Harrison [59], Summary Cache [61], etc. [83] gives an overview of these approaches.

Both Hierarchical Caching and Distributed Caching allow the sharing and coordination of cache state among multiple communicating caches to improve system performance and scalability, and fall into a broader category known as “Cooperative Caching”. There are two important aspects
investigated in most Cooperative Caching research: finding nearby caches which hold the cached content and coordinating the caches while making storage decisions. The first aspect has been widely studied and there are a lot of approaches such as the ICP [57] in Harvest project, Summary Cache [61] and Adaptive Web Caching [55]. Taking Summary Cache as an example, it exploits a directory-based scheme where each proxy cache maintains a directory that summaries the cached content on other proxy caches. Upon a “cache miss”, the proxy cache can forward the request to the nearby proxy cache (or the origin server) based on the directory information. The work focusing on the second aspect, coordinated cache placement, includes [63, 64], both of which investigated optimal algorithms for coordinated cache placement problem and proposed a greedy algorithm within a factor (1.1 - 1.5 for [63] and 14 for [64]) of optimal for practical use. [67] investigated the impact of Cooperative Caching on network performance and scalability and provided a quantitative evaluation based on a two-pronged approach: trace-based analysis and analytic modelling. The results reveal that Cooperative Caching has performance benefits only within limited population bounds (e.g., for small-organization proxies with populations ranging from 200 to 2000 users, Cooperative Caching can increase the average hit rate on a single proxy by 9% to 17%; however, for large-organization proxies with population larger than 20000, the improvement is between 3.3% and 3.7%). The results also imply that the increased latency of inter-proxy communication in the wide area network overshadows the benefits of Distributed Caching, advocating for a hierarchical variant.

2.3.2 Peer-To-Peer Caching

In contrast to a traditional client-server content delivery network, a peer-to-peer (P2P) content delivery aims at utilizing a client’s spare resources to provide a high performance caching mechanism, presenting an interesting research area for academic and industrial communities [46, 44]. Relying on a peer-to-peer routing protocol, the participant can join or leave the system at any time and any place, providing a highly self-organizing content delivery network.

An on-going research project in the Secure Computer Systems (SCS) group of NYU, named Coral [46], focuses on building a peer-to-peer content delivery network, which takes advantage of the unutilized upload capacity of users to re-distributed contents, resulting in a scalable, high performance network. Nodes that volunteer to run Coral automatically replicate contents when users access them and publish these replicated contents through Coral by simply prepending a pseudo-hostname to objects’ URLs; a peer-to-peer DNS layer is applied to transparently redirect requests to nearby participating cache nodes. Coral relies on a distributed sloppy hash table (DSHT) [45, 47] for distributed DNS lookup which is designed to avoid hot spots where large numbers of (key, value) pairs in DSHT have the same key by guaranteeing that even if every node of a DSHT repeatedly stores the same key, the rate of store requests at the most heavily-loaded node is only logarithmic in the total number of nodes.

Although a peer-to-peer content delivery network can achieve high performance by serving the requests from the nearby cache nodes, the performance could drop dramatically if the number of volunteers is small. Furthermore, this approach has a higher connection time due to P2P DNS lookup and request redirection, and does not provide QoS assurance. Finally, the security concern of caching possibly sensitive content on untrusted volunteer nodes remains a big issue.

2.3.3 Web Caching for Dynamic Content

Web servers have been used traditionally to hold pre-generated static content. Yet there exists another sort of information where the documents or components of documents are generated dy-
namically upon receiving client request. Such content is referred to dynamic content, and includes examples such as rotating advertising banners, CGI scripts, ASP pages, etc.

Although the exploitation of dynamic content alleviates the effort of web site maintenance and results in richer visitor experience for users, existing web architectures have inherent inefficiencies in the delivery of such content. The main challenge of dynamic content caching is how to guarantee the freshness, consistency and accuracy of the content in the cache. Traditional caching approaches, such as [38, 41, 39, 42, 43], can only handle static content and dodge the dynamic content caching problem by marking them as “non-cacheable”. However, recent studies of proxy cache effectiveness in [67] show that the fraction of all requests for dynamic content can amount to as much as 50%, and is likely to keep increasing with time.

Studies show that although the content to serve user requests might change over large time duration, this content remains unchanged for certain shorter time periods. And even when it changes, there are portions that may remain unchanged. Most dynamic content caching approaches take advantage of this fact to introduce the concept of “freshness” for the content stored in the cache and apply an invalidation-based or update-based consistency mechanism. [65, 69] proposed a fragment-based approach for dynamic content caching. The approach relies on the observation that dynamic web pages can be composed based on simpler entities, known as fragments. Fragments typically represent parts of Web pages which change together such that when a change to underlying data occurs which affects several Web pages, the fragments affected by the change can be easily identified. Hence, the content in the cache can be updated on the level of fragments, not Web pages, and the performance could be improved significantly. In such an approach, users or content providers need to specify how Web pages are composed from fragments by creating templates in a markup language. These templates are parsed to determine inclusion relationships among fragments and Web pages, which is represented as an object dependence graph (ODG). The ODGs are used to determine how changes should be propagated throughout the Web after one or more fragments change.

In this section, we discuss several dynamic content caching approaches.

Active Caching

Active Caching [60] was first proposed in the WisWeb project at the University of Wisconsin, Madison. It targets the web personalization problem. A recent study [66] of HTTP traces from a large ISP reveals that about 30% of all user requests carry cookies, the HTTP header elements typically indicating that a request be personalized.

The key component in Active Caching is an applet: a piece of specialized code provided by the origin server which is attached to the response content. Both content and the associated applets are stored in the cache and upon a subsequent request, the cache can execute applets which customizes the document to serve the request.

The major drawback of this mechanism is that the content providers have to relinquish control over part of the application logic, preventing it from being widely exploited.

CONCA

CONCA [70] addresses two major issues for web content access: dynamic content caching and “on-the-move” access for mobile users, and points out that there does not exist a solution which can independently solve both of these problems.

Following the context of fragment-based approach discussed above, the CONCA design proposes two key enablers which reside on the server side and the client side respectively to provide
information about content structure and user content preferences. The server side enabler assumes that the content supplied by server can be associated with a “document template” which defines both the structure and form of content. The client side enabler, called Personal Assistant, identifies the kinds of devices the user has access to, the templates for each of these devices, transcoding information about the original content and converted content, and additional information such as consistency linkage between these two kinds of contents. By exposing such information on both server side and client side, CONCA automatically bridges the semantic gap between the two to enhance the effectiveness of proxy caches. CONCA applies a distributed client-side proxy caching architecture to support caching of dynamic, transcoded content. For nomadic users, CONCA makes use of a home cache which maintains per-user state persistently, and allows the persistent state to be recreated on another cache as the user moves.

The main weakness of CONCA is that it requires significant communication between proxy caches for exchanging cache states. Also, it needs administration to set up two enablers on both server and client sides. Furthermore, it is not clear that how much the nomadic user can benefit from the use of the home cache since the communication between two caches will introduce additional overhead.

### Database-Driven Caching and Semantic Caching

A large number of web servers consist of two parts: an application server and a back-end database, where the application server processes the user request and generates the corresponding query against the back-end database; the back-end database is responsible for data fetching and management.

Recent work on improving the performance on such systems by caching portions of the back-end database includes [68, 73, 74, 72, 76]. These approaches usually rely on a query wrapper and a query matcher to intercept the request at the cache, and based on the query matching result, either run a corresponding query against a local storage or invoke a remote query for the origin database. The differences between these approaches are the consistency mechanism and the storage organization of the cache. The consistency mechanism used in [68, 73, 74, 72] is invalidation-based, while in [76] an update-based consistency mechanism is employed. For the storage organization, [68, 73, 74] use an unstructured, form-based mechanism to represent the cached data which results in a redundant storage problem; [72, 76] uses a stand-alone database which contains only a portion of the origin database.

The different strategies for storage organization discussed above can be characterized into two categories: Page Caching and Tuple Caching. Page Caching is widely used in operating and database systems, assuming each query can be broken down to the level of requests for individual pages. The data in the proxy cache are organized as individual pages and upon a “cache miss”, the proxy can retrieve the particular pages from the origin server on behalf of the end user. Tuple Caching maintains the data in the cache in terms of individual tuples, resulting in a higher flexibility compared with Page Caching. The tuples in the cache are indexed using their primary keys.

Although these strategies work when there is an explicit back-end database, they can not be used in situations such as web-based retrieval systems where requests can still be viewed as queries against a database but there may not be a physical database storing the data required for the response. In such cases, the cache contains the responses which may be dynamically generated for the requests, indexed by the query contained in the request. Page Caching fails because the query used in such information system is keyword-based and the data organization at the servers is completely hidden from the clients (and is not necessarily a page-based organization). Tuple Caching fails because the request contains a filled search form that describes a query instead of a
primary key and furthermore, it can not inform the server about the already cached tuples in order to reduce response size.

To overcome these deficiencies, [89, 90, 92, 93, 94] proposed Semantic Caching. In Semantic Caching, data in the cache are managed as a collection of Semantic Regions. A Semantic Region is a set of tuples that are defined and adjusted dynamically based on the queries posed at the client. The usage of Semantic Regions not only addresses the limitation of Page Caching in page-based data organization, but also resolves the indexing problem for Tuple Caching. Furthermore, it avoids the high storage overheads in Tuple Caching strategy which has to maintain the replacement information on a per-tuple basis. Upon a query being posted, the query is split into two parts: a probe query and a remainder query. The former is used to retrieve the portion of the answer available in the local cache and the latter is used to retrieve the missing data from the origin server [89].

IBM WebSphere Edge Server and Akamai EdgeSuite

There are several commercial approaches to Dynamic Content Caching. IBM WebSphere Edge Server [48] provides a mechanism to offload application components, such as servlets, Java Server Pages, Enterprise Beans, to the edge server in the network. The edge server, acting as an application-server proxy, can handle some dynamic content requests locally and forward the others to the origin server. The major disadvantage for this scheme is that for data-intensive applications, where the data have to be fetched from the origin server, the improvement of web performance and scalability is quite limited.

Akamai EdgeSuite [50] leverages IBM WebSphere to allow Akamai’s customers to distribute their Web application workload to the network. Akamai relies heavily on Edge Side Includes (ESI), a markup language that breaks Web pages into fragments [65, 69] with a profile describing the ability to cache items. Fragments may be labeled as cacheable for different time scales, such as days, minutes or seconds. Upon request, the fragments are assembled into an HTML page at the edge and only those fragments deemed impossible to cache are retrieved from the origin server. The main drawback of ESI is the substantial overhead required to re-tag pages to identify cacheable content.

3 Motivating Example

In this section, we describe the motivating example for our work and its challenges.

3.1 MapPoint Web Services

The motivating example for our work is Microsoft’s MapPoint Web Services [3]. Microsoft’s MapPoint Web Service is an XML-based Web service that enables developers to integrate location-based services, such as maps, driving directions and proximity searches, into their applications and business processes. The client usually requests the service using SOAP messages, following the specification in the WSDL document for MapPoint Web Services. In our motivating example, we focus on the map retrieval aspect of the MapPoint service, which mainly involves the “GetMap” operation.

Figure 1 shows a typical usage scenario of MapPoint Web Services. In this scenario, the web server that hosts MapPoint Web Services resides in Seattle, associated with an extremely large maps database. There are several network intermediaries across the wide area network, which act as SOAP Routers in relaying SOAP messages. These intermediaries are registered at an UDDI
service point. The client finds the nearby intermediary via UDDI and requests the service by sending a SOAP message to that intermediary. The SOAP messages might travel one or multiple intermediate SOAP Routers before arriving at the server and similarly, the responses might follow the same network path back to the client. The network is heterogeneous, i.e., the linkages have different bandwidths and latencies. Clients can use a variety of devices, such as PCs, laptops, PDAs and mobile phones, to request service. More importantly, client requests have associated QoS requirements. For example, a client desiring online driving directions may interact with the service while requiring that responses take no more than a certain latency. Similarly, another client wanting to understand distribution of roads in a geographic area may care about more detailed maps and less about the latency of request.

The MapPoint Web Service usage possesses an important feature: the client requests exhibit locality over space and time, at different granularities. For example, in the city granularity, users in Chicago may primarily request maps around the city of Chicago while those in New York may mainly request maps around the city of New York. Similarly, such locality exists in the state, the coast granularity, etc. Another characteristic of the MapPoint Web Service is the “polymorphism” of the operation. Taking “GetMap” as an example, Figure 1 shows that it can take three different kinds of parameters as input: ViewByHeightWidth, ViewByScale and ViewByBoundingRectangle, associated with another parameter indicating the image size shown on screen. The ViewByHeightWidth determines a map by a center point (in latitude and longitude) and the height and width of maps in DistanceUnit (in miles or kilometers); the ViewByScale determines a map by a center point and a map scale; and the ViewByBoundingRectangle determines a map by the latitude and longitude coordinates that represent the southwest and northeast corners of a minimum bounding rectangle. These three different representations are interchangeable by applying a simple computation to convert one to any other. In other words, there could be different requests that represent the same query against the map database.

To improve the scalability and performance of such an application, a possible strategy is to use service replication, i.e., deploying the service replicas properly at the intermediaries. The service replication should involve both application distribution and data offloading to achieve better performance. Due to the extremely high volume of the original map database, it is infeasible to offload the whole database to the intermediaries, instead, only a portion of the origin database can be offloaded. To be effective, such a portion of the origin database should have high access rate, i.e., represent locality in the usage of the database.

There are several challenges that need to be overcome for this service replication-based strategy to be practically relevant:
• The infrastructure should be able to provide QoS assurance to the end clients.
• The infrastructure should be able to resolve multi-representation of SOAP requests.
• The service replication usually occurs in a large-scale, heterogeneous wide area network, where the cost of the service replication is not free. Hence, the service replication needs to be associated with a cost-benefit model.
• Since the changes of service usage patterns and network conditions usually happen in a small time scale, the infrastructure needs to be able to adapt to such changes automatically.
• The locality of service usage can exist at different levels of granularity, with respect to both network region and database space. It is impossible (or very hard) to efficiently detect such locality from a centralized point of view. Therefore, a more effective and distributed mechanism is needed.

To our best knowledge, there does not exist an approach, neither of component-based service integration nor involving web caching and replication, which can address the above issues entirely.

To understand where existing approaches fall short, one can view the problem of building a scalable web services infrastructure along three dimensions:

• The first dimension describes how complex is the cacheable content (content complexity). Possible points along this dimension include static content, dynamic content, and more sophisticated structured content (e.g., semantic regions of the database).
• The second dimension describes the “reach” of the infrastructure, and ranges from stand-alone proxy caching to more complex networks of cooperating caches.
• The third dimension describes the guarantees provided by such an infrastructure, and span “best-effort” assurances at one end and more complex QoS assurances involving request latency and quality at the other.

Our motivating example falls within a region defined by the most sophisticated functionality along each of these dimensions — working with structured content, involving a network of cooperating network entities, and requiring QoS assurances. Along this latter dimension in particular, very few approaches (an exception includes [82]) have looked at providing any QoS assurances over wide-area networks.

3.2 Proposed Approach

To address the challenges presented in our motivating example, we propose a novel approach which takes advantage of the concept of “Semantic Region” from Semantic Caching and applies it to model web service usage patterns (in both computation and data usage) in order to support the highest level of content complexity while providing QoS assurances for client requests. Our approach makes use of a collection of locations distributed across the network, which act as SOAP Routers in message relaying. These locations are organized according to a certain type of network topology (e.g., a hierarchy), and are responsible for the detection of service usage patterns and their changes over space and time, as well as the monitoring of changes in network conditions. This information is used to determine an optimal strategy for service replica placement in order to guarantee the QoS metrics, which in turn ensure the scalability and high performance of the infrastructure. To address the issue of multi-representation of SOAP requests, our approach exploits XSLT to resolve the polymorphism in requests.
4 Architecture

In this section, we discuss the architecture of WSDAF. Inspired by the motivating example of MapPoint Web Services, the preliminary design and experiments of WSDAF focus on data intensive Web Services, where requests mainly involve data retrieval and there is a back-end database sitting behind the web server. We believe that the results can be applied to computation intensive Web Services as well.

4.1 Preliminaries

To clarify our subsequent presentation, we introduce some commonly used terms:

Terminology

- **Server**: An application which accepts connections from clients, receives client requests and sends out responses. In WSDAF, Server refers to a XML Web Services server; both client requests and server responses are SOAP messages, transmitted via HTTP.

- **Origin Server**: The server on which the original content resides or is to be created.

- **Client**: An application which establishes connections to make service request. A client can be a web browser or a program that implements web services APIs published via WSDL. The connection can be connected either to an origin web server or to a cache server, more specifically, WSDAF Router in WSDAF.

- **WSDAF Router**: Like a Proxy server, a WSDAF Router is an intermediary system that acts as both a server and a client for the purpose of making requests, on behalf of other clients. Unlike Proxy server, WSDAF router is basically a Soap Router implementing WS-Routing protocol and WS-Referral protocol, which can relay client requests based on preexisting referral configurations. WSDAF Router extends the functionality of Soap Router to act as proxy server.

- **Cell**: A representation of a portion of the database at the origin server. Assuming that the database is multi-attribute and that the attributes are numerical or alphabetical rangeable, a cell represents a hyper-region, i.e. hyper-rectangle, in a multi-attribute space. Inspired by the concept of a Semantic Region in Semantic Caching, WSDAF uses Cells as the granularity at which to model database usage patterns by grouping user requests into different Cells.

- **CellTree**: A tree-structured representation of the origin database (Figure 2). Assuming the attributes of the origin database have both lower and upper bounds, the whole data space defined by the database can be split into smaller hyper-rectangles, i.e., individual Cells.

- **Cell Replica**: As we discussed before, service replication in a data-intensive Web Services infrastructure has to involve data offloading. A Cell Replica is a portion of the origin database which is cached on the WSDAF Router, defined by a Cell.

- **Hit Rate**: The number of user requests that hit in a Cell during a unit time. Hit Rate is used in WSDAF to determine Cell usage patterns: a Cell with high Hit Rate represents locality in database usage.
4.2 WSDAF Architecture

WSDAF leverages the advantages of XML and Web Services to provide a scalable, QoS-assured Web Services infrastructure that adapts to changes of network characteristics. Figure 3 shows an overview of the WSDAF architecture.

The WSDAF Routers are organized into a hierarchical architecture where each intermediate node is an extended SOAP Router [14, 20, 21] that acts as a Web Services Proxy by intercepting user SOAP requests for a web service and making a decision on whether it should pass the requests to its parent or it can serve them from a local cache. The cache on a WSDAF Router contains a set of hyper-regions defined by Cells which are portions of the origin database. Scalability is achieved by both offloading Web Services to and transparent replication of data on WSDAF Routers; adaptability is achieved by dynamically monitoring user usage patterns and network performance; and QoS-assurance is achieved by enforcing QoS metrics at each level of the WSDAF Router network.
4.2.1 Design Overview

As Figure 3 shows, WSDAF is a three-tier Web Services infrastructure: client ⇔ router ⇔ server, where the middle tier is a WSDAF Router Layer consisting of multiple WSDAF Routers which are organized in a hierarchical architecture.

This hierarchy can be configured either automatically or manually by setting up the WS-Referral relationship between WSDAF Routers. At this stage, WSDAF relies on manual configuration.

WSDAF Router Layer embodies almost all of the functionality provided by WSDAF, such as message relaying, request serving on behalf of the origin server, usage pattern detection, network conditions monitoring and service replica deployment, etc. To accomplish such functionality, WSDAF Router needs necessary information about a web service, including the WSDL document of the web service, meta-data information about the back-end database from which the service fetches data, and implementation logic which indicates how a service processes the inputs to generate queries against the database.

The WSDL document of the web service is needed because WSDAF Router relies on it to retrieve the information about service API (operation)†, inputs for the operation, and service bounding information, which is used to construct service proxy and relay user requests. The meta-data information is needed by WSDAF Router because one of the goals of WSDAF Router is to discover service usage patterns by monitoring service requests to infer the database utilization. The service implementation logic is needed by WSDAF Router because there is a semantic gap between a SOAP message, representing a request for service, and a query against the database. To be able to intercept and monitor user requests, WSDAF Router has to have knowledge about how to transform a SOAP message into a database query.

With the above information present, WSDAF Router is able to provide its functionality via three kinds of functional components.

**Service Component** WSDAF Router needs to know the necessary information about a web service.

†Although a web service can provide multiple functionalities through multiple operations and stores the descriptions of these operations and the related bounding information in a WSDL document, to simplify our description, we assume that a web service provides only one operation.
service on which it operates. With such information, WSDAF Router can construct the service proxy and publish its service to a UDDI service point. The WebService Registry component provides this functionality.

Core Components The Core Components provide: (1) A mapping mechanism to map a service request onto a query over the back-end database (SoapRequest Translator). (2) A data space representation of the back-end database such that WSDAF Router can model the usage patterns over the database. This is achieved by the design of our internal data structure, CellTree, which is associated with monitoring components to perform request monitoring and usage pattern modelling. In order to capture the usage pattern at small granularities, our data structure allows dynamical growth to integrate information at different levels of resolution. (3) A monitoring mechanism to monitor the user requests and detect usage patterns among these requests with the aid of the internal associated data structure (Access Monitor). (4) A mechanism that can monitor changes of network properties (Performance Monitor). (5) A mechanism to manage the service replication, more specifically, the offloading of data for data-intensive Web Services (Caching Managers).

Policy Components The Policy Components are components which deal with the policies for QoS assurance enforcement, internal resources management strategies, cost of deployment of service replicas over the WAN, and communication protocols between WSDAF Routers. These policies act as guidelines for the Core Components to ensure QoS metrics, exchange internal states, determine an optimal strategy for service replica deployment and improve resource usage efficiency.

4.2.2 WebService Registry

WSDAF Router is designed to support request relaying and service replication for generic web services. To invoke the functionality for a particular service, the user, usually an administrator, needs to register related information about the web service via the WebService Registry component. The information about a web service includes its WSDL document, an XML Stylesheet used by XSLT to transform SOAP requests, and meta-data information about the back-end database.

Based on the supplied meta-data information, WSDAF Router constructs a CellTree object to represent the data space defined by the back-end database and associates this CellTree object with SOAPAction of operation in WSDL document. Also, with the description of operation in the WSDL document, WSDAF Router can create a proxy which can serve user requests on behalf of the origin server in the case that requests hit in the local cache, or relay the requests otherwise.

The registration information is stored in a persistent table (as a file on the local disk), indexed by SOAPAction of operation supported by a web service.

4.2.3 SoapRequest Translator

There is a semantic gap between XML Web Service request and the corresponding query against the service database. Typically, the query arguments need to be extracted from the parameters of the request. In general, it is not possible to build a generic convertor that will work with different database designs. The alternative of requiring designers to specify arbitrary convertor functionality is associated with security concerns, because these convertors need to execute on the SOAP routers.

WSDAF implements a compromise solution. It permits designers to specify conversion functionality in a safe language at the cost of placing restriction on the kinds of operations that can be performed. Specifically, request conversion in WSDAF are specified as XSLT templates, permitting
their application in a generic fashion. The WSDAF convertors extract the query by applying the specified XSLT transformation on the XML document representing the service request.

XSLT is a safe language because it does not provide access to operating system functionality such as file system, I/O and memory operations. Most of the build-in XSLT functions inherit XPath functions and only provide the functionality of XML Node Set, string, number and boolean operations. However, XSLT supports third party extensions, such as EXSLT [17] and Java, via extension elements and extension functions. If these extensions do not use proper security procedures, the security of the host could be compromised.

4.2.4 Access Monitor

In order to successfully employ the service replication algorithm, WSDAF requires a monitoring mechanism which dynamically monitors information about usage patterns and resource utilization.

The Access Monitor component summarizes user access information. Using information registered in the WebService Registry, the Access Monitor is able to construct a CellTree object which represents the data space defined by the back-end database. This CellTree object is used to collect user access information.

Initially, the CellTree contains a single cell, the root of the tree, which represents the entire data space. The cell keeps track of the number of requests that hit in its region. When the hit count reaches a predefined threshold, \( cellCapacity \), the cell is split \(^1\) into a set of disjoint sub-cells, forming a tree (see Figure 5). Each sub-cell is initialized with the same hit count by evenly dividing the hit count of the parent. For subsequent requests, the access information is collected only at the sub-cell level. This splitting procedure continues until the depth of CellTree reaches a predefined maximum value, \( maxSplitDegree \), which defines the smallest granularity of data space at the back-end database. For example, in our motivating example, a larger value of CellTree depth might indicate that WSDAF Router is monitoring map requests at city level, while a smaller value might refer to a state level granularity.

The user access information collected in the CellTree structure is exchanged between WSDAF Routers. To prevent ambiguity, each database portion, represented by a Cell, is denoted by a unique Cell name. When a Cell is split, the child Cells are sorted into a list according to their positions in the region of the parent Cell and given names using the indices in the list. For example, the root Cell has a name “0”, its children have names as “0.0”, “0.1”, etc.

The access information of each cell is reset periodically to reflect the changes of usage patterns. More precisely, the hit count of a cell stands for the number of requests that have hit in this cell since the last reset time. One can also imagine using other summarization techniques, such as exponential averaging.

In our splitting procedure, each sub-cell receives an equal share of hit count of the parent, assuming that accesses at the parent were uniformly distributed within the region. However, such an assumption may not be correct. To catch situations where some portions of a cell are accessed more frequently than others, we adopt the following strategy. Assuming that the access patterns do not change quickly, the cell splitting procedure can be enhanced by a Cell Hit-count Adjustment procedure to provide a better estimate for the sub-cell shares of parent’s hit count, which is invoked only once for each newly split sub-cell. We discuss this adjustment procedure in detail in Section 5.1.1.

The final issue about the Access Monitor is how to count a hit in a Cell. The simple case is when the request area (volume) falls completely into the region defined by a cell. In cases where the request area spans several cells, we increase the hit count of each of the overlapping cells.

\(^1\)For simplicity, the splitting is done using binary division on each dimension of data space.
4.2.5 Performance Monitor

To enable WSDAF to be adaptive to changes in network characteristics, WSDAF Routers have to be able to detect these changes on the fly. The Performance Monitor component is designed for this purpose.

Performance Monitor monitors the network latency for a web service request either between two WSDAF Routers or between a WSDAF Router and the origin server on a per-cell basis. More specifically, Performance Monitor measures the average response time (ART) for all of the requests that hit in a cell. A WSDAF Router can infer that if service performance has been impaired due to congestion on the path or poor performance on an upstream node by detecting large fluctuations in these values.

Performance Monitor also maintains the network latencies between two nodes. This information, combined with a predefined QoS metric requirement value, helps determine an optimal strategy for service replica deployment.

4.2.6 Caching Managers

In data intensive Web Services, service replication mainly involves data offloading. Therefore, following the terminology of Web Caching, we refer to data offloading as caching, and the service replica deployment problem as the cache placement problem.

WSDAF applies a QoS-metric driven caching scheme: no data is cached until some intermediate nodes detect that they can not guarantee the QoS metrics. Since user access information is tracked on the Cell level, caching also happens on the Cell level.

Cache Repository

Data in a WSDAF Router cache is stored persistently in a local stand-alone database semantically consistent with the back-end database, using the meta-data information registered in WebService Registry. Contents in the WSDAF Router cache are described by indices which consists of queries defining the regions.
Cache Placement

Most web caching approaches aim at caching contents with high hit probability on front-end proxy servers. Theoretically, end-user performance improves as more content is cached and as the front-end nodes are selected closer to the clients. However, front-end proxy servers are usually resource constrained, preventing them from holding a high volume of contents from the back-end database in their local caches. Furthermore, data offloading across the heterogeneous WAN is not free.

WSDAF applies a Cost-Benefit Model to solve the cache placement problem. The cost is associated with the distance between the replica and the origin server and the benefit is associated with the amount of reduced response time. A distributed algorithm is applied to find an optimal cache placement with minimum cost in order to ensure the QoS metric.

Cache Consistency

Data consistency is an important issue in web caching approaches. The data in local caches may get stale due to changes to contents occurring at the origin server because of update operations (Update, Delete and Insert).

WSDAF applies an update-based propagation protocol to ensure that cached data is consistent with the origin server. In WSDAF, the execution of updates operations are only allowed at the origin server. Each WSDAF Router simply passes these requests on to its parent without executing them against the local cache. The origin server propagates the committed changes to all of the WSDAF Routers along the hierarchy. Each WSDAF Router, upon receiving such a propagation, identifies the affected cells in the local cache and applies the changes only on these cells.

This Cache Consistency mechanism presumes slowly changing data, which is typical of most web environments.

Cache Replacement

The Cache Replacement component ensures efficient management of limited space on the WSDAF Router. Specifically, it periodically evicts from the cache the following type of data: the contents of a cell which has been unfrequently accessed recently.

The Cache Replacement component applied a cache replacement policy to determine what should be replaced. However, the definition of a cell being ineffectively used is not straightforward due to variations in user access behaviors. Using the access information from Access Monitor, the Cache Replacement component defines a cell as being ineffectively used in a gradual manner (see Section 5.2.2).

Once the replaceable cells are identified, the Cache Replacement component removes the contents defined by these cells and updates the corresponding cache indices to reflect the change.

5 Algorithms, Protocols and Policies

In this section, we discuss the algorithms, protocols and policies used in designing and implementing WSDAF. In Section 5.1.2, we define the general Cache Placement Problem, and prove that it is an NP-hard problem; we also give a definition of a restricted Cache Placement Problem and provide both centralized and distributed algorithms for it. In Section 5.2, we present the communication protocols which are designed to exchange information between WSDAF Routers and the origin server and discuss the policies enforced in WSDAF to ensure QoS metrics and internal resource management on WSDAF Routers.
5.1 Algorithms

5.1.1 Cell Splitting: Hit-count Adjustment Algorithm

In section 4.2.4, we mentioned the Cell Splitting procedure and the motivation for hit count adjustment. We discuss this algorithm and its policy in detail in this subsection.

In an n-dimensional space, a cell can be essentially split into $2^n$ sub-cells. Each sub-cell is initialized with a hit count ($initHit$), evenly divided up $cellCapacity$, a predefined value indicating that a cell should be split (at split time, the hit count of a parent cell is equal to $cellCapacity$). Each sub-cell then keeps track of the hit count of subsequent requests, ($newHit$), from the moment that the parent was split. So, the hit count of a sub-cell consists of two parts: $initHit$ and $newHit$.

The adjustment is triggered by one of the following events: (1) a global Cell Hit-count Resetting procedure, which is invoked periodically; (2) one of the sub-cells hit count reaches $cellCapacity$ and has to be split. When either these events happen, each sub-cell computes the proportion of its $newHit$ to the sum of $newHits$ of all of the sub-cells belonging to the same parent, and uses that proportion to adjust its $initHit$ value. Once the adjustment is done, hit counting resumes as in the normal case.

The pseudo code for this algorithm is in Appendix A.

5.1.2 Cache Placement Problem and Algorithms

Cache Placement Problem

The goal of WSDAF is to provide a QoS-assured web service to the end clients. In the following discussion, we focus on one QoS metric: web service response time. A common way to reduce response time is to cache selected web contents on the front-end proxy servers. Since cache misses are possible, the QoS assurance is usually expressed as a threshold on the average response time for clients or average service time for servers.

To satisfy these requirements, two problems need to be addressed: what to cache and where to place the cache? The solution to the first problem is rather simple: the contents that receive hit with high probability should be cached. The second problem, without the cost concerns, can be simply solved by placing the cache on the proxy servers that are closest to end clients. However, if we take both cost concerns and QoS assurance into consideration, the problems addressed above turn out to be hard to solve.

Although the structure of the WSDAF Router hierarchy could in general be a directed acyclic graph (DAG), we start our investigation with a tree-based hierarchy and show that the general cache placement problem even in such cases is an NP-hard problem. Hence, we place additional restrictions on the cache placement problem and prove that the restricted cache placement problem can be solved in polynomial time by giving both centralized and distributed algorithms. Our next goal is to extend these algorithms to a super-tree-based hierarchy where the intermediate node is a cluster of WSDAF Routers such that each WSDAF Router can have multiple parents and WSDAF Routers in the same cluster can cooperate with each other. We believe that such super-tree-based hierarchy corresponds more closely to existing Internet architecture.

**General Cache Placement Problem (GCPP)**

*Given a tree-based WSDAF Router hierarchy, a database that has been split into multiple partitions, a distribution of client accesses at the entry level Routers, and a QoS threshold on the average service time, find a min-cost cache placement strategy such that the overall infrastructure satisfies the QoS requirement, i.e., the average service time of entry level Router falls below the QoS threshold.*
Cost-Benefit Model

In order to reason about the GCPP, we introduce a Cost-Benefit Model:

We assume that: (1) the clients access web services only via the entry level WSDAF Routers; (2) the QoS assurance of average response time to clients is provided by a threshold of average service time at entry level Routers; (3) the bandwidth of the link between two WSDAF Routers is unlimited; (4) the CPU power of a WSDAF Router is unlimited. The last two assumptions simplify our reasoning by ensuring that once a partition of the database is placed on a Router, it benefits the nodes in that Router’s subtree, and that the cached partition does not affect service time for requests that do not hit in this partition. The reason for this is that after a partition of the database is cached on an intermediate Router, the requests that hit in this partition have shorter service time and hence the user access rate for this partition may increase, resulting in more bandwidth and CPU power being consumed. If the user access rates for all of the database partitions are fixed, then the last two assumptions are not necessary.

Now, examining the cost of creating a database partition on a router node, we can express this as:

\[
\text{Cost} = c_1 \times \text{Vol} + c_2 \times \text{Dist}
\]

where \(\text{Vol}\) is the size (in bytes) of this partition of the database and \(\text{Dist}\) is the distance between the origin server and the Router where this partition is placed; \(c_1, c_2\) are predefined constants.

The service time (ST) of a WSDAF Router for a single request can be broken down into three parts: request processing time (\(T_p\)) at the Router, round-trip network delay between the Router and its parent (\(T_{\text{lat}}\)), and the service time of the parent (\(T_{\text{parent}}\)):

\[
ST = T_p + T_{\text{lat}} + T_{\text{parent}}
\]

Assume that the processing time (\(T_p\)) at local Router is negligible (\(T_p << T_{\text{lat}} + T_{\text{parent}}\)), then:

\[
ST \simeq T_{\text{lat}} + T_{\text{parent}}
\]

The average service time of a Router is computed by taking the average of the service times for all requests processed by this Router. Assuming that the number of requests is \(r\), and the service time for request \(i\) is \(ST_i\):

\[
\text{AST} = \frac{\sum_{i=1}^{r} ST_i}{r}
\]

Once a partition of the database is created at this Router, for those requests that hit in this partition, \(T_{\text{lat}}\) and \(T_{\text{parent}}\) become 0. Assuming the probability of a request hitting in the cache is \(P_{\text{hit}}\), we have:

\[
ST' = P_{\text{hit}} \times T_p + (1 - P_{\text{hit}}) \times (T_{\text{lat}} + T_{\text{parent}})
\]

Since \(T_p\) is negligible,

\[
ST' \simeq (1 - P_{\text{hit}}) \times (T_{\text{lat}} + T_{\text{parent}}) = (1 - P_{\text{hit}}) \times ST
\]
The benefit of cache placement is measured by the amount of time saved in serving the requests. If a database partition is placed at an entry level Router, the benefit can be calculated as:

\[ \text{Benefit} = \text{reqs} \times (\text{AST} - \text{AST}') = P_{\text{hit}} \times \text{reqs} \times \text{AST} \quad (6) \]

This equation can also be used to compute the benefit of placing a database partition at an intermediate Router \( R \) based on the following observations: (1) Only the nodes in the subtree of that intermediate Router can benefit from this cached database partition; (2) The network characteristics in this subtree do not change, i.e., the network latencies and bandwidths of the links between Routers in this subtree remain the same; and (3) The client request pattern remains the same (assuming that the service requests from individual clients are not dictated by their service times, but other client-specific factors).

If the entry level Routers in this subtree are \( R_1, R_2, \ldots, R_n \), Router \( R_i \) sees \( r_i \) number of requests, and the average service time before and after cache placement are \( \text{AST}_i \) and \( \text{AST}_i' \) respectively, for any \( i \in [1, n] \), then the benefit can be calculated as:

\[ \text{Benefit} = \frac{\sum_{i=1}^{n} (r_i \times (\text{AST}_i - \text{AST}_i'))}{\sum_{i=1}^{n} r_i} \]

Notice that the number of requests at an intermediate Router \( R \) equals the sum of the requests at each entry level Router in its subtree, and \( (\text{AST}_i - \text{AST}_i') = (\text{AST} - \text{AST}') \) for all \( i \in [1, n] \), where \( \text{AST} \) and \( \text{AST}' \) are the average service times of Router \( R \) before and after cache placement. Combining Equations (2), (3), (4) and (5), the above equation reduces to Equation (6).

Figure 6 shows a simple example of cache placement in WSDAF. In this example, the infrastructure QoS metric for the average response time is 5 units, and the network linkages as well as their round-trip latencies are shown in the figure. The algorithm first identifies that the unsatisfied entry-level Routers are \( r_0, r_1 \) and \( r_2 \) and then chooses \( r_0 \) and \( r_3 \) to place the caches. (An alternative placement strategy is to select \( r_0, r_1 \) and \( r_2 \) as replicas, but has higher cost.)

![Figure 6: A simple example of Cache Placement](image-url)
The GCPP is NP-hard

We simplify GCPP by considering a special WSDAF Router hierarchy: the chain-based hierarchy (Figure 7). In the chain-based hierarchy, each intermediate Router has only one child. We show that even this simplified GCPP is NP-hard.

![Figure 7: Chain-based WSDAF Router Architecture](image)

Suppose that the chain consists of \( n \) Routers where \( R_1 \) is the entry level Router and \( R_n \) is the root Router, the database has been split into \( m \) disjointed portions, and the cost for caching data portion \( P_j \) on Router \( R_i \) is \( c_{ij} \). Assume that service time for a request is only determined by the network latencies of the links between Routers, denote the round-trip latency between \( R_1 \) and \( R_i \) as \( l_i \), consequently, \( l_1 \) equals 0. Given a client request distribution over the \( m \) portions, \( r_1, r_2, \ldots, r_m \), a QoS threshold \( t \) of average service time, and using the following terms: \( \text{reqs} = \sum_{j=1}^{m} r_j \) and \( T = t \times \text{reqs} \), the simplified GCPP can be formulated as:

\[
\begin{align*}
\text{minimize:} & \quad \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} \cdot x_{ij} \\
\text{subject to:} & \quad \sum_{j=1}^{m} r_j \sum_{i=1}^{n} l_i \cdot x_{ij} \leq T \\
& \quad \sum_{i=1}^{n} x_{ij} \leq 1, j = 1, \ldots, m \\
\text{where:} & \quad x_{ij} = \begin{cases} 
1, \text{if database portion } j \text{ is cached at Router } R_i \\
0, \text{otherwise}
\end{cases}
\end{align*}
\]

Let the average service time on \( R_1 \) before cache placement be \( A\text{ST}_{R_1} \), denote \( C = A\text{ST}_{R_1} \times \text{reqs} - T \) as the total amount of time that should be reduced to satisfy the QoS assurances. Let \( L \) be the round-trip network latency between the root Router \( R_n \) and the entry level Router \( R_1 \) such that \( l_i' = L - l_i \) is the amount of time that could be saved if a cache is placed at Router \( R_i \). The simplified GCPP is reduced to finding the min-cost cache placement in the chain such that the amount of saved time at entry Router \( R_1 \) is at least \( C \):
minimize: \( \sum_{i=1}^{n} \sum_{j=1}^{m} c_{ij} x_{ij} \)

subject to: \\
\( \sum_{j=1}^{m} r_{j} \sum_{i=1}^{n} l'_{i} x_{ij} \geq C \) \\
\( \sum_{i=1}^{n} x_{ij} \leq 1, j \in [1, m] \)

This is the minimized version of Multi Choice Knapsack Problem (MCKP) \[96\], which is known to be NP-hard.

**Restricted Cache Placement Problem**

We have proven that GCPP is an NP-hard problem. To understand whether there exists a different formulation of the problem amenable to an efficient solution, we enforce a restriction on the GCPP and prove that this version of the GCPP, named Restricted Cache Placement Problem (RCPP), is a polynomial problem by providing both centralized and distributed algorithms for it.

**Restricted Cache Placement Problem (RCPP) Statement**

Given a tree-based WSDAF Router hierarchy, a database that has been split into multiple partitions, a distribution of client accesses at the entry level Routers, and a QoS threshold of average service time, find a min-cost cache placement strategy such that for any entry level Router, the average service time of each database partition, considered separately, remains within the QoS threshold.

RCPP differs from GCPP in imposing a per-partition QoS requirement, which may increase the cost over that required by GCPP.

Because the database is split into multiple disjointed partitions, RCPP can be broken into a set of independent subproblems, each subproblem works on only one database partition.

**Claim:** A min-cost cache placement strategy for RCPP is equal to the union of min-cost cache placement strategies for RCPP’s subproblems, where each subproblem works on only one database partition

**Proof:**

First, by assumption, the database partitions are independent from one another. So a cache created for a partition will not affect the performance of another.

Second, the union of the min-cost cache placement strategies for subproblems provides a solution for RCPP because it ensures that the average service time of the entry level Router for each database partition remains within the QoS threshold.

Finally, the union of the min-cost cache placement strategies for subproblems is a min-cost cache placement strategy for the RCPP. Assume the union of min-cost cache placement for subproblems is \( S = \bigcup_{i} s_i \), where each \( s_i \) is a min-cost cache placement strategy for a subproblem. Suppose there exists another cache placement strategy \( S' \) which has less cost. Obviously, \( S' \) can be divided into a set of disjoint cache placements \( \{s'_i\} \), each of which contains only the cache placement for one database partition. In other words, each \( s'_i \) is a solution for a subproblem. Then, there exists at least one \( s'_k \),
Cost($s'_k$) < Cost($s_k$). This contradicts the definition of $s_k$.

To simplify the description of algorithms to subproblems of the RCPP, without loss of generality, we assume that all of our entry level Routers are unsatisfied, i.e., for a specific database partition, the average service time of each entry level Router exceeds the QoS threshold. This assumption does not affect the cache placement by the observation that already satisfied Routers do not require any cache placement in the Router hierarchy to reduce their average service times. Thus, we can reduce a general WSDAF Router tree to a tree where the leaf nodes only consists of unsatisfied Routers.

**Router Tree Reduction Algorithm**

- Apply a depth-first search on the Router tree
- Mark a node as “unsatisfied” if: (1) the node is an entry level Router and its average service time exceeds the QoS threshold; or (2) any of its children are marked as “unsatisfied”
- Remove all of the unmarked nodes

The last question we have to answer is how to find a min-cost cache placement strategy for a subproblem in polynomial time.

**(I). Centralized Algorithm**

**Centralized Algorithm: Min-Cost cache placement for a subproblem**

- Rank entry level Routers by the extent of their discontent, i.e., the amount of time that the average service time exceeds the QoS threshold
- Start with the most unsatisfied entry level Router, traverse along the path from this Router to root, find the highest intermediate Router where the cache can be placed in order to satisfy the entry level Router
- Remove all the nodes in the subtree rooted at this intermediate Router (except the intermediate Router)
- Repeat this algorithm until there are no unsatisfied entry level Routers left

**Claim**: The generated cache placement strategy is optimal

**Proof**: The algorithm is a greedy algorithm.

Suppose the entry level Routers are $r_1, \ldots, r_m$, in order of rank based on the extent of discontent from high to low. Suppose the router chosen by the algorithm to place a cache on in order to satisfy $r_1$ is $R$. We claim that any min-cost cache placement strategy should include $R$ as one of the nodes to place a cache on. The proof follows immediately by the following observations:
1. By the definition of cost (Equation 1), because the database portion being cached is fixed, the only variant that affects the cost is the distance.

2. A cache placed on a node residing in \( R \)'s subtree has a cost not less than the one placed at \( R \).

3. A cache placed on a node residing outside of \( R \)'s subtree can not satisfy \( r_1 \).

By induction, the algorithm generates an optimal solution for a subproblem.

**Complexity**

The algorithm runs in \( O(n \times D \times M) \), where \( n \) is the number of database partitions, \( D \) is the depth of Router tree and \( M \) is the number of unsatisfied entry level Routers.

**(II). Distributed Algorithm**

<table>
<thead>
<tr>
<th>Distributed Algorithm: Min-Cost cache placement for a subproblem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Each Router maintains the round-trip latency of the link between its parent and itself, denoted as ( t_{lat} ).</td>
</tr>
<tr>
<td>• Each unsatisfied entry level Router computes the amount of time, denoted as ( t_{dec} ), which needs to be reduced in serving a single request, such that the average service time of this Router remains within the QoS threshold ( Q ): ( t_{dec} = AST - Q ).</td>
</tr>
<tr>
<td>• Each unsatisfied entry level Router compares its ( t_{lat} ) with ( t_{dec} ). If ( t_{lat} &gt; t_{dec} ) (which means this Router can not be satisfied by placing a cache on its parent), the Router issues a cache placement request to the origin server and then sends a satisfactory message to its parent; otherwise, it sends an unsatisfactory message to its parent, with the portion of the required reduction time that must be satisfied by the parent (equal to ( t_{dec} - t_{lat} )).</td>
</tr>
<tr>
<td>• The intermediate Router collects the messages, satisfactory or unsatisfactory, from all of its children. If all the messages are satisfactory, the intermediate router sends a satisfaction message to its parent; otherwise, it sets its ( t_{dec} ) as the smallest required reduced time in the unsatisfactory messages from its children and follows the same steps as the entry level Router does</td>
</tr>
<tr>
<td>• Upon receiving all of the children satisfaction messages, the root Router issues a cache placement request if there exists any unsatisfactory message</td>
</tr>
</tbody>
</table>

**Claim:** The generated cache placement solution is optimal

**Proof:**

The claim is proven if any optimal cache placement solution should include all of the nodes that issue cache placement requests to the origin server in the algorithm. We prove it using mathematical induction.

Since our algorithm works in bottom-up order where the lower level Routers send satisfaction messages to their parents, starting at the lowest level, we refer a round to a
step of the algorithm where Routers at a particular level of the tree send their satisfaction messages. Therefore, round 0 indicates that the algorithm works at the entry level Routers. Suppose the depth of the Router tree is \( D \), we have:

At round 0, the algorithm checks entry level Routers and issues a cache placement request if and only if the \( t_{\text{lat}} \) of the Router is greater than its required reduced time \( t_{\text{dec}} \). These entry level Routers that issued cache placement request should be in the optimal cache placement solution.

Suppose at round \( k, k < D \), the nodes that issued cache placement request according to the algorithm are in the optimal cache placement solution.

At round \( k+1 \), if there is an intermediate Router \( R \) which is chosen by the algorithm to place the cache, then there exists an unsatisfied entry level Router \( r \), which has the following properties:

1. the round-trip latency between \( r \) and \( R \) is not greater than the reduced time required by \( r \)
2. the round-trip latency between \( r \) and \( R \)'s parent is greater than the reduced time required by \( r \)
3. no cache has been placed on any node on the path from \( r \) to \( R \)
4. any node residing in \( R \)'s subtree, which has been chosen in an earlier round to place the cache, is in the optimal cache placement solution (follows immediately by the induction)

To satisfy entry level Router \( r \), the optimal cache placement solution has to include \( R \) as one of the nodes holding the cache, which means that the nodes chosen to place caches up to the tree level \( D - k - 1 \) should be included in the optimal cache placement.

By mathematical induction, the algorithm generates an optimal solution.

**Complexity**

The algorithm will terminate after round \( D \), where \( D \) is the depth of the Router tree. The number of messages that need to be sent for communication between Routers is at most \( 2N \) where \( N \) is the number of nodes in the Router tree.

### 5.2 Protocols and Policies

#### 5.2.1 Performance Monitoring Protocol

Figure 8 shows the message exchange between neighboring Routers in the WSDAF Router tree. The parent router measures the service time (\( ST \)) in serving a request by computing the amount of time from the moment when the request comes in up to the moment when the response is ready, and sends this information as part of the response message. The child router uses this information to compute the round-trip time latency (\( T_{\text{lat}} \)) between the two, which it then passes back to the parent as part of the next forwarded request, \( T_{\text{lat}} = ST_{\text{child}} - ST_{\text{parent}} \).
Each WSDAF Router computes the average service time (AST) on a per-Cell basis: (1) each Cell keeps track of the number of requests (hitCount) that have hit in this Cell and the average service time in serving those requests (2) upon receiving a new request that hits in this Cell, the Router first increase the Cell hitCount by 1, then measures the service time (ST) in serving this request and updates the average service time of this Cell using the equation:

\[
AST = \frac{(hitCount - 1) \times AST + ST}{hitCount}
\]

**Communication protocol**

- WSDAF Router measures the service time for each incoming request and sends this information along with the response back to the downstream Router.
- WSDAF Router computes the RTT between itself and its parent and sends this information along with the next forwarded request to its parent.

![Performance Monitoring Protocol](image)

**Figure 8: Performance Monitoring Protocol**

### 5.2.2 Cache Replacement Policy

The Cache Replacement Policy is responsible for identifying replaceable Cells in the cache. We have discussed what kind of Cells that are considered for replacement in Section 4.2.6. Here, we discuss how to identify such replaceable Cells.

- Each Cell in the cache is associated with an internal counter, *cellUtil*, whose value indicates the access frequency for this Cell. There are also two predefined values, named *highFreq* and *lowFreq*, which are used to clarify the cell behavior.

- Initially (when the cell is just replicated), *cellUtil* is set to an initial value indicating that the access frequency of this cell is high.

- Periodically, the hit count of this Cell is compared with *highFreq* and *lowFreq*: if hitCount \(\geq\) highFreq, *cellUtil* is set to the initial value; if hitCount \(\leq\) lowFreq, *cellUtil* is decremented by 1.
• Once cellUtil reaches 0, this Cell is identified as being ineffectively used and hence is replaceable.

This policy identifies the ineffectively used cells in a gradual manner. In this way, it avoids mis-replacement of cells which might have a low access frequency temporarily due to a fluctuation in user access behaviors.

6 Implementation

WSDAF Router is built on top of Microsoft’s ASP.NET Framework 1.1 [26] and Microsoft Web Services Enhancement Package (WSE), version 2.0 [25], which provides support for the WS-Routing and WS-Referral protocols. As a SOAP Router, WSDAF Router implements the ASP.NET IHTTPHandler interface with extensions for WS-Routing and WS-Referral protocols to provide SOAP routing functionality. Although WSE has provided a routing implementation, named Microsoft.Web.Services.Routing.RoutingHandler, we chose to develop our own for compatibility, flexibility, and efficiency considerations. For example, the WSE implementation hides details of headers of HTTP requests, which might contain useful information such as the client’s authentication tokens. Hiding such information from the router might cause the re-routed request to be rejected by the server due to security concerns. Although WSE can provide a security mechanism by making use of WS-Security [22], which is an enhancement to SOAP messaging, it requires code modification on client side and hence is less desirable.

WSDAF Router extends ASP.NET IHTTPHandler to provide request monitoring and data offloading functionality. The Access Monitor, Performance Monitor components as well as Caching Managers are dynamically constructed from the registered information in the WebService Registry. The cache storage, though can make use of a local stand-alone database, is currently implemented as a structured directory system.

WSDAF Router is registered with Microsoft Internet Information Services (IIS) 6.0 ([9]) as a customized HTTP handler. IIS is responsible for dispatching incoming requests to different registered HTTP handlers according to the type of the request and invoking the corresponding module to process requests.

The WebService Registry component is implemented as an ASP WebApplication that interoperates with the WSDAF Router. The application provides a form for the administrator to register information about the web service and the related database. The registered information is stored in a file on disk and is used by WSDAF Router to initialize the monitoring and caching components.

The source code of WSDAF Router is written in C#. The total length, as of this writing, is about 3,900 lines.

7 Performance Evaluation

In this section, we present an experimental study of the performance benefit of the WSDAF infrastructure. The experiments were conducted on a simulated system, with network topology and link characteristics such as delay and bandwidth modelled using Click, a modular software router. User requests were simulated by running threads on a 32-CPU cluster, sending requests to a 8-node WSDAF Router tree.
7.1 Experiment Setup

7.1.1 Experiment Environment

Figure 9 shows the sketch of our experiment environment. There are ten machines used in our simulated system:

- an ASA PC (CPU: AMD Athlon™ XP 1800+ 1.5GHz; Memory: 512 MB)
- a 32-CPU computing cluster (Chicken) (CPU: AMD Athlon™ XP 1800+ 1.5GHz; Memory: 512 MB)
- eight HP PCs (CPU: Pentium-4 2.4GHz; Memory: 512MB)

The ASA PC is installed with Click and acts as a IP router to simulate a network shown in Figure 9. The HP PCs are installed with Microsoft ASP.NET Framework 1.1 and WSE 2.0, running on Windows Server 2003. These eight machines are configured to act as WSDAF Routers, \( R_1, \ldots, R_8 \), except that \( R_8 \) also acts as the origin server. The Chicken, which can be viewed as 32 individual machines, \( C_0, \ldots, C_{31} \), which share the same filesystem, is used to generate the client workload.

The simulated network consists of eight sub-nets, \( Net_1, \ldots, Net_8 \). Each sub-net consists of one WSDAF Router and four clients, with the internal network bandwidth and latency being set to 10 Mbits and 3 msec respectively. The network bandwidth and latency between sub-nets are configured to be 3 Mbits and 10 msec respectively.

7.1.2 Evaluation Methodology

**WSDAF Router**

The web service evaluated in our experiment is Microsoft’s MapPoint Web Service. Because the MapPoint web site restricts the number of requests processed per unit time for an evaluation account (the only feasible option), it is difficult to do experiments with heavy load on the official server. Therefore, we decided not to use the real server hosted by Microsoft, instead, we created a proxy on \( R_8 \), which provides the same interface as the WSDL document published by MapPoint, and can
simulate the MapPoint server to serve requests. A simple structured directory system is used to act as a maps database and provide the contents needed in generation of MapPoint response: a file represents a whole map of North America at a specific resolution.

The WSDAF Routers are configured to form a tree-based hierarchy as shown in Figure 9. The QoS threshold of average service time of each Router is set to 500 msec. We also set the granularity of database partitioning, i.e., the depth of CellTree, to be 6. For a map resolution at 50000 : 1, the smallest granularity corresponds to a rectangle on the earth’s surface with size about 50miles × 50miles: this approximately corresponds to maps at city level. The size of such a smallest portion at resolution 50000 : 1 is about 11 MB, computed based on the measurement of the size of a map returned from the MapPoint site.

We do not cache all of the portions of database that are unsatisfied, instead, we only cache those that have an access rate higher than 1.5 requests per second. The Cell access rates are reset by the Cell Hit-count Resetting procedure, which is invoked every 300 seconds.

Client
Client requests are sent to the closest Router (the Router within the same sub-net) where they may be relayed to an upper level Router or processed locally. To model spatial locality, the 32 clients are divided into 8 groups as shown in Figure 9, each group targets a region defined by a center point and a width in the database. The clients that belong to the same group generate workload that randomly requests maps within the group region. Furthermore, the group center points are selected randomly from another region where the center point is predefined, e.g., the center point of database, and the width is provided at runtime.

Two parameters are used to control the spatial locality of requests: \( \alpha \) and \( \beta \). Both of them are ratios with respect to the width of the region defined by the whole database. \( \alpha \) is used to define a region from which the group center points can be selected; \( \beta \) is used to define the region for individual groups. By tuning the values of \( \alpha \) and \( \beta \), we can model different levels of locality for user requests.

In our experiment, clients only requests maps with resolution 50000 : 1, and the size of each map is set to 400 × 400 pixels. The size of the SOAP message for client requests is 4 KB and the size of the response message is 34 KB. These numbers come from measurement of real requests and responses from the official Microsoft MapPoint web site.

There are two types of workload that are generated. (1) Synchronized Workload: the clients of all eight groups start off together and repeatedly send requests to the closest Routers. The experiment lasts 2 hours. (2) Staggered Workload: the starting time of groups is staggered over the experiment, so that at most two groups are active at any time.

To ensure a constant load (within reasonable limits) on the Router, the client uses multiple threads to maintain the number of requests being sent out per unit time. Our experiment used 5 threads, each sending out a request every second. As long as the request service was below a second, each client would generate requests at a fixed rate of 5 requests per second.

Before presenting the performance of WSDAF under load, we would like to show the basic measurement of its performance with a single client and no caches. We measure the response times for clients residing in different sub-nets: \( Net_1 \), \( Net_5 \) and \( Net_8 \), respectively. These results are shown in Figure 10.

### 7.2 Performance Under Load

Since the network is symmetric, we only present the performance of one representative group at each level: \( Net_1, Net_5 \), and \( Net_8 \). We show changes in the response time seen by clients, and
discuss the cache placement on the Routers. Notice that we measure average response time over a 1-second period: in the figure, the Y-axis value for a point on the curve is the average response time of requests whose responses arrive in the previous one second.

7.2.1 Synchronized Workload

Case 1: Large \( \alpha \), large \( \beta \)

In this experiment, \( \alpha \) is set to be 0.1, representing a sparse distribution of group center points; \( \beta \) is set to be 0.1, representing a relatively low locality for requests sent by clients that reside in the same group. The number of Cells in each group region can be computed as follows: \( 2^6 \times 2^6 \times (0.1 \times 2) \times (0.1 \times 2) = 164 \).

The results (Figure 11) show that all of the clients suffer long response times: by average about 5.5 sec for clients in \( Net_1 \), \( Net_5 \) and about 2.8 sec for clients in \( Net_8 \). The reasons for the long response times are: (1) the server is overloaded (the CPU utilization gets above 80% on \( R_8 \)); (2) the network is saturated, especially for the link between \( Net_8 \) and \( Net_5 \), which causes the clients in \( Net_5 \) seeing almost the same response time as those in \( Net_1 \).

Due to the long response time, the number of requests sent to Router \( R_1 \) during each resetting period (300 sec) can be computed:

\[
20 \frac{\text{req}}{\text{sec}} \times 300 \text{ sec} / 5.5 \frac{\text{sec}}{\text{req}} = 1091 \text{ req}
\]

Hence, the number of requests sent to Router \( R_5 \) during each update period is about \( 3 \times 1091 \), where one third of requests are passed by \( R_1 \), one third of requests are passed by \( R_2 \), and the rest come from the clients in \( Net_5 \). Even if all the clients in these three groups were accessing the same group region, the average hit count during each update period for a Cell is about \( 3 \times 1091 / 164 = 20.5 \). Therefore, no Cell can reach the hot-spot Caching threshold we have set for the experiment.

Case 2: Small \( \alpha \), large \( \beta \)

In this experiment, \( \alpha \) is set to be 0.01, representing a dense distribution of group centers; \( \beta \) is set to be 0.1, representing a relatively low locality for requests sent by clients that reside in the same group.
With this setting, the regions of different groups overlap with each other. However, because the value of $\beta$ is still large, the locality of requests observed by Router $R_1$ is still very low, and even for $R_5$, locality is not large enough to make any Cell become a Hot-Spot.

Figure 12 shows the performance for this case, exhibiting essentially the same performance as Case 1.

**Case 3: Large $\alpha$, small $\beta$**

In this experiment, $\alpha$ is set to be 0.1, representing a sparse distribution of group centers; $\beta$ is set to be 0.01, representing a high locality for requests sent by clients that reside in the same group.

Table 1 shows the access distributions of client requests in different groups.

<table>
<thead>
<tr>
<th>Cell</th>
<th>Fraction</th>
<th>Cached locally</th>
<th>Cached on parent</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clients in Net$_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0.3.3.3.2.0</td>
<td>0.473</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>0.0.3.3.3.2.1</td>
<td>0.527</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clients in Net$_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.1.2.2.2.2.1</td>
<td>0.487</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.1.2.2.2.3.0</td>
<td>0.513</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Clients in Net$_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.2.1.1.1.3.0</td>
<td>0.445</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>0.2.1.1.1.3.2</td>
<td>0.555</td>
<td>yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Figure 13 shows that for requests of clients in Net$_1$, a cache of Cell 0.0.3.3.3.2.1 is first placed on $R_5$ at time 300 sec; and 300 seconds later, two caches for Cell 0.0.3.3.3.2.0/1 are placed on $R_1$ since $R_1$ finds that the average response time in both Cells still exceed the QoS threshold. After these two Cells being cached on $R_1$, the average response time observed by clients in Net$_1$ reduces.
Figure 12: Performance under Synchronized Workload with small $\alpha$, large $\beta$

to about 100 msec.

For Router $R_5$, besides two caches for Cells accessed by the local clients being placed on $R_5$, there are two other caches that also get placed: Cell 0.0.3.3.2.1 for clients in $Net_1$ and Cell 0.1.2.2.2.2.1 for clients in $Net_2$. The latter two caches turn out to be not really helpful to clients in $Net_1$ and $Net_2$ and are removed shortly after the corresponding Cells are cached on $R_1$ and $R_2$ respectively. The reason for this is as follows (take Cell 0.0.3.3.2.1 as an example): before time 300 sec, $R_1$ found out that the average response time of Cell 0.0.3.3.2.1 exceeded the QoS threshold and the round-trip network latency between $R_1$ and $R_5$ is smaller than the QoS threshold (about 200 ∼ 300 msec), therefore it sent an unsatisfied message to $R_5$ and asked $R_5$ to cache this Cell. However, after 0.0.3.3.2.1 was cached on $R_5$, the service time for requests hitting in this Cell was reduced significantly and hence the access rate for this Cell increased (this is because the client was configured to send out requests at the rate of 5 req/sec, but due to the long service time in the case that no cache is placed, the actual requesting rate from a client ends up being smaller than 5 req/sec). As a result: (1) $R_5$ had to serve more requests than before; (2) the queueing time for these requests at $R_5$ increased; and (3) the bandwidth between $R_5$ and $R_1$ had to be shared among more response messages. Overall, the service time of $R_5$ for requests from $R_1$ increased to a point where $R_1$ found that its average service time of Cell 0.0.3.3.2.1 exceeded the QoS threshold and had to issue another request to have this Cell cached locally. Ideally, a model that can precisely predict the effect of a cache placement is desirable. Unfortunately, without the knowledge about the client access behaviors (not only what is observed, but is desired) as well as the network characteristics, such a ideal model is not feasible. In section 7.2.4, we discuss several policies that try to compensate for the mistake in cache placement due to this information shortage.

In Figure 13, the response time line of $C_{28}$ goes steady as late as after time 1440 sec. The reason for this is that $R_8$ also acts as the origin server and therefore needs to provide the copies of Cells to be offloaded. The data downloading consumes both the bandwidth of $Net_8$ and CPU power of $R_8$.

Case 4: Small $\alpha$, small $\beta$

In this experiment, $\alpha$ is set to be 0.01, representing a dense distribution of group centers; $\beta$ is set to be 0.01, representing a high locality for requests sent by clients that reside in the same group. The result is that the requests of clients in all of the groups fall in the same four Cells, but
different group clients have different fractions of requests in accessing these four Cells (Table 2).

Table 2: **Access Distribution: small $\alpha$, small $\beta$** (for synchronized workload)

<table>
<thead>
<tr>
<th>Cell</th>
<th>Fraction</th>
<th>Cached locally</th>
<th>Cached on parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clients in $Net_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0.3.3.3.3.3</td>
<td>0.736</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.1.2.2.2.2.2</td>
<td>0.026</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>0.2.1.1.1.1.1</td>
<td>0.230</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.3.0.0.0.0.0</td>
<td>0.008</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Clients in $Net_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0.3.3.3.3.3</td>
<td>0.137</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.1.2.2.2.2.2</td>
<td>0.560</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.2.1.1.1.1.1</td>
<td>0.049</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>0.3.0.0.0.0.0</td>
<td>0.214</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Clients in $Net_5$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.0.3.3.3.3.3</td>
<td>0.020</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>0.1.2.2.2.2.2</td>
<td>0.006</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>0.2.1.1.1.1.1</td>
<td>0.739</td>
<td>yes</td>
<td>N/A</td>
</tr>
<tr>
<td>0.3.0.0.0.0.0</td>
<td>0.235</td>
<td>yes</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The experiment shows that two caches are placed on $R_1$ for Cell 0.0.3.3.3.3.3 and 0.2.1.1.1.1.1. On the other hand, all four of the Cells have caches placed on $R_5$. Notice that clients in $Net_1$ have very low access locality on the two Cells not being cached. Therefore, the response time line of $C_0$ is sawtooth after time 600 sec, which is caused by the requests that belong to uncached Cells that have to be served by $R_5$ and hence have longer response time.

The glitch in the response time line for $C_2$ (Figure 14) between time 2100 sec and 2280 sec is because of the placement of a cache on $R_7$. This cache had been placed on $R_7$ early in this experiment but was removed due to its temporary low access rate.

### 7.2.2 Staggered Workload

In the following experiments, the 8 groups of clients do not start sending requests at the same time. The order in which groups sending requests $Net_1$, $Net_2$, $Net_3$, $Net_4$, $Net_5$, $Net_6$, $Net_7$, $Net_8$, with 900 seconds delay between any two consecutive groups. Each group sends requests for 1200 seconds and terminates.

We only show the performance of two consecutive groups, $Net_1$ and $Net_2$.

**Case 1: Large $\alpha$, large $\beta$**

**Case 2: Small $\alpha$, large $\beta$**

As we discussed in the Performance under Synchronized Workload, no caches get placed for a large value of $\beta$. Figures 15 and 16 show that during the first 900 sec, $C_0$ has a flat response time line. But the line goes up after time 900 sec as soon as the second group joins in. Similarly, the response time line of $C_4$ goes down at time 1200 sec because the first group terminates.
Figure 13: Performance under Synchronized Workload with large $\alpha$, small $\beta$
Figure 14: Performance under Synchronized Workload with small $\alpha$, small $\beta$
Figure 15: Performance under Staggered Workload with large $\alpha$, large $\beta$

Figure 16: Performance under Staggered Workload with small $\alpha$, large $\beta$
Case 3: Large $\alpha$, small $\beta$

In this case, clients in $Net_1$ and $Net_2$ access different Cells, with the fractions shown in Table 1.

Figure 17 shows that the caches for the two Cells accessed by clients in $Net_1$ are placed on $R_5$ at about time 300 sec. However, the congestion which occurred on the link between $R_8$ and $R_5$ moves to the link between $R_5$ and $R_1$ after the cache placement; therefore, the service time improvement observed by $R_1$ is not obvious. (Notice that each client is able to send 5 requests per second, which means $R_5$ has to serve 20 requests every second. Each response has size 34 KB, hence there is 680 KB of data that needs to be transmitted through the link every second, exceeding the link capacity.) Therefore, at time 600 sec, these Cells have to be placed at $R_1$ in order to keep the average service time within the QoS threshold. After that point, the response time line of clients in $Net_1$ remains steady at about 60 $\sim$ 70 msec.

At the time that clients in $Net_2$ start sending requests, the requests from clients in $Net_1$ are served by $R_1$, so the response time lines of $C_0$ and $C_4$ do not go up as they did in Case 1 and 2. Clients in $Net_2$ have similar performance as those in $Net_1$ since they have almost the same access behaviors.

Case 4: Small $\alpha$, small $\beta$

In this case, the Cell access fractions for clients in $Net_1$ and $Net_2$ are shown in Table 2.

Figure 18 shows that two Cells, 0.0.3.3.3.3 and 0.2.1.1.1.1.1, which have higher access fraction, are cached at $R_5$ at time 300 sec. Then as the congestion moves to the link between $R_5$ and $R_1$, $R_1$ has to request cache placement for Cell 0.0.3.3.3.3 at time 600 sec, which reduces the response time and in turn increases client access rate. At time 900 sec, $R_1$ finds it has to request another cache placement for Cell 0.2.1.1.1.1.1, which further reduces the response time. Notice that requests that hit in the other two Cells (0.1.2.2.2.2.2 and 0.3.0.0.0.0.0) have to be forwarded to $R_5$ and then to $R_8$ to get served, which causes the fluctuation of the response time even after the caches have been placed.

The clients in $Net_2$ have similar performance as those in $Net_1$ except that three Cells (0.0.3.3.3.3, 0.1.2.2.2.2.2 and 0.3.0.0.0.0.0) instead of two are cached on $R_2$ at time 600 and 900 sec, respectively. Only Cell 0.2.1.1.1.1.1 does not need to be cached due to its low access fraction.
Figure 18: Performance under Staggered Workload with small $\alpha$, small $\beta$

7.2.3 Performance of Origin Server

The content offloading can not only reduce the end-clients’ response time, but can also improve the performance of the origin server. We show the CPU performance of $R_8$ in the experiment with Synchronized Workload for large $\alpha$, small $\beta$.

Figure 19 shows that the CPU utilization drops from 80% to 43% when the Cells are cached on the intermediate Routers; and after the cache placement finishes, the CPU utilization remains steady at about 31%.

Figure 19 also shows that the concurrent requests for $R_8$ drops after the caches have been placed. This is because most of the requests from the clients residing outside of Net$_8$ will be served by the caches placed on the corresponding Routers.

Figure 19: Performance of WSDAF Router $R_8$
7.2.4 Performance impact of caching policies

Request repression policy

The request repression policy is designed to repress new requests hitting in a cell which is being replicated on the WSDAF Router. In a saturated network, the cell replication could worsen the network load temporarily due to the high volume of content being offloaded. There are two choices for handling requests hitting in such cells: (1) pass the request to the Router’s parent or (2) repress the request until the replication is finished and then serve it locally. Figure 20 shows the difference of performance between forwarding or repressing the request. Notice that in the case that request is forwarded, the replication procedure takes a longer time to finish. The figure reveals that the request repression policy can result in better performance.

Cell replication broadcasting policy

During our experiments, we found out that when an intermediate WSDAF Router \( R \) replicates a cell, its child or parent Router might request replication of the same cell later, even if their performance on that cell would be under the QoS threshold after replica creation on \( R \). The reason this happens is because the child or parent Router does not get notified about the fact that replication is happening on \( R \), or what its effects are likely to be, and hence can not update their access information immediately. The cell replication broadcasting policy is designed to prevent such redundant replication.

Cache replacement policy adjustment

As we discussed in Section 5.2.2, the cache replacement policy uses two predefined value to measure the cell utilization. However, in our previous experiments, the values of \( \text{highFreq} \) and \( \text{lowFreq} \) were set to the same value, which causes some replicated cells to be removed. These cells which are temporarily not accessed frequently but will resume high access rates later. The adjustment for this policy is to set \( \text{lowFreq} \) equal to half of \( \text{highFreq} \).

Figure 21 shows that the effect of applying the adjustment of cache replacement policy: there is a sharp increase of response time for \( C_{28} \) at time 2100 (figure (a)) due to a downstream Router.
dropping its useful replica, while in figure (b), where the replacement does not happen, the performance of $C_{28}$ is steady after time 1100.

**Cache replication policy adjustment**

In the previous experiments, all cell replication requests were processed by the origin server. We adjust this policy by allowing a lower level WSDAF Router to populate its cache from its upstream Router if the corresponding cell has been replicated there. Such an adjustment can improve the utilization of the network.

Figure 22 (a) shows that WSDAF Router $R_1$ in $Net_1$ takes about 150 seconds to replicate a cell before applying the above adjustment; but in (b), it takes only 30 seconds to pull the same copy of cell content from its parent, $R_5$. 
8 Discussion

In this proposal, we have discussed the challenges presented in building a scalable, high-performance Web Services infrastructure and argued that existing approaches can not address these challenges entirely. To overcome these challenges, we proposed a novel approach that leverages the concept of Semantic Regions to model web service usage patterns on a set of preexisting network entities. These entities are organized into a hierarchy, and use service usage information to determine an optimal strategy for service replica placement to maintain infrastructure QoS assurances. We have presented an implementation of our approach, WSDAF, which is used as a prototype for investigation. The performance evaluation of WSDAF on a simulated system shows that WSDAF is able to detect the existence and changes of locality among the client requests over space and time, using an internal dynamically adaptive data structure. The results also show that the designed communicating protocols and policies, as well as the distributed algorithm, work correctly to permit WSDAF to adapt to changes of service usage patterns and network characteristics by offloading a portion of services with high locality to appropriate locations in an automatic fashion, with minimal cost. Our evaluation reveals that WSDAF has the potential to provide a scalable, high-performance Web Services infrastructure.

To help realize this potential in practical settings, our plan of future work consists of the following three stages:

Stage 1 We plan to evaluate our architecture on realistic network workloads (or simulated ones based on realistic web traces). Ideally, we would like to evaluate our architecture on commercial web sites, such as Microsoft’s MapPoint web site or TerraServer web site. However, getting access to these commercial web sites seems an impossible mission, and even getting access to realistic web traces seems unlikely: we have interacted with Jim Gray at Microsoft in this regard, but have not been successful so far.

As an alternative suggested by Dr. Gray, we are investigating web traces from the public SkyServer data portal. SkyServer is a public website that presents data from the Sloan Digital Sky Survey [5], a project that aims to make a map of a large part of the universe. The public logs on this data portal contain 37M web site accesses and 2.7M SQL queries, available for retrieval or online analysis.

We intend to first identify the spatial and temporal properties in SkyServer web traces and then use this information to evaluate our techniques. We will also consider looking for published reports about the characteristics of web traffic on commercial web sites from other sources, and using these reports to model service usage patterns in our workloads.

Stage 2 We plan to generalize the network topology in our framework. The current topology used by WSDAF is a tree-based hierarchy. We will extend such hierarchy to a super-tree-based one where each intermediate node is a cluster of WSDAF Routers, and each WSDAF Router can have multiple parents which reside in the same cluster. We speculate that this hyper-tree-based hierarchy corresponds more closely to existing Internet architecture.

As part of the work in this stage, we will first evolve the routing protocol used by the architecture to accommodate the situation that a lower level Router could have multiple parents. We will also need to extend the distributed algorithm for service replica placement since the incoming requests on a Router could be distributed to different parents, and similarly modify the monitoring component since the responses to a Router could come from different parents.
As an enhancement, we might also consider a more fancy protocol that can intelligently determine the best routing path using the information about linkage performance between Routers as well as client privileges (e.g. group right priorities).

**Stage 3** We plan to extend the functionality of our internal data structure: Cell. The Cell structure is used by our monitoring component to keep track of service usage patterns as well as service performance in an dynamically adaptive fashion. Although it is in principle possible to use a database to store client requests and apply “slicing” and “dicing” techniques to detect locality in service usage, it is costly to do so since the number of client requests could be tremendous and the database operations could take a lot of time to process. The Cell structure does not attempt to store the client requests, instead, it stores their statistical information and maintains this information in an adaptive way by updating it every time window. However, the current implementation of the Cell structure is limited to storing the access information of the last window.

We will refine the Cell structure to keep track of the access information in multiple time windows. For space concerns, it will be more efficient to store the variation between two consecutive windows, rather than to store the detailed information in each window. Such information about the service usage in multiple windows can be viewed as time series data, hence, recent developed data mining techniques for time series data, e.g. [98], can be applied to discover useful information such as correlations among usage patterns to provide better prediction.

Our work has made two assumptions: (1) we assume a trusted relationship across the administrative domains that participate in injecting code onto WSDAF Routers, and (2) we assume the existence of a shared web ontology to permit service sharing and reuse across domains. Although there are several interesting research problems in these fields (security infrastructures, sandbox environments or semantic-web like issues), we will use existing “best” solutions for these problems and focus our research on the issues identified above.
Appendices

A  Cell Splitting: Hit-Number Adjustment Algorithm

Pseudo code for Cell Hit-Number Adjustment:

initialize:
    n is the data space dimension;
    \( N := 2^n \); // the number of sub-cells

procedure Cell_HitNumber_Adjustment ( )
{
    if ( this.hasChild )
    {
        foreach ( Cell cell in this.childs )
        {
            cell.Cell_HitNumber_Adjustment( );
        }
    }
    else
    {
        var totalNewHit := 0; // the total hit-number of all sub-cells since parent splitting
        foreach ( Cell cell in this.parent.childs )
        {
            totalNewHit += cell.newHit;
        }
        initHit := initHit * N * newHit/totalNewHit;
    }
}
References


49


51


[80] X. Gu, K. Nahrstedt, R. Chang and C. Ward. QoS-Assured Service Composition in Managed Service Overlay Networks. 23rd Int’l Conf. on Distributed Computing Systems (ICDCS), Providence, Rhode Island, May 2003


[87] M. Kim, S. Lam and D. Lee. Optimal Distribution Tree for Internet Streaming Media. 23rd Int’l Conf. on Distributed Computing Systems (ICDCS), Providence, Rhode Island, May 2003


