Information Systems Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

SOA Performance Enhancement Through XML Fragment Caching

Anindya Datta, Kaushik Dutta, Qianhui Liang, Debra VanderMeer,

To cite this article:
Anindya Datta, Kaushik Dutta, Qianhui Liang, Debra VanderMeer, (2012) SOA Performance Enhancement Through XML Fragment Caching. Information Systems Research 23(2):505-535. https://doi.org/10.1287/isre.1110.0368

Information Systems Research
Vol. 23, No. 2, June 2012, pp. 505–535
ISSN 1047-7047 (print) | ISSN 1526-5536 (online)

http://dx.doi.org/10.1287/isre.1110.0368

SOA Performance Enhancement Through XML Fragment Caching

Anindya Datta
Department of Information Systems, National University of Singapore, Singapore, datta@comp.nus.edu.sg

Kaushik Dutta
Department of Information Systems, School of Computing, National University of Singapore, Singapore, duttak@nus.edu.sg

Qianhui Liang
Hewlett Packard Labs, Fusionopolis, Singapore, qianhui.liang@hp.com

Debra VanderMeer
Department of Decision Sciences and Information Systems, College of Business, Florida International University, Miami, Florida 33199, vanderd@fiu.edu

Organizations are increasingly choosing to implement service-oriented architectures to integrate distributed, loosely coupled applications. These architectures are implemented as services, which typically use XML-based messaging to communicate between service consumers and service providers across enterprise networks. We propose a scheme for caching fragments of service response messages to improve performance and service quality in service-oriented architectures. In our fragment caching scheme, we decompose responses into smaller fragments such that reusable components can be identified and cached in the XML routers of an XML overlay network within an enterprise network. Such caching mitigates processing requirements on providers and moves content closer to users, thus reducing bandwidth requirements on the network as well as improving service times. We describe the system architecture and caching algorithm details for our caching scheme, develop an analysis of the expected benefits of our scheme, and present the results of both simulation and case study-based experiments to show the validity and performance improvements provided by our caching scheme. Our simulation experimental results show an up to 60% reduction in bandwidth consumption and up to 50% response time improvement. Further, our case study experiments demonstrate that when there is no resource bottleneck, the cache-enabled case reduces average response times by 40%–50% and increases throughput by 150% compared to the no-cache and full message caching cases. In experiments contrasting fragment caching and full message caching, we found that full message caching provides benefits when the number of possible unique responses is low while the benefits of fragment caching increase as the number of possible unique responses increases. These experimental results clearly demonstrate the benefits of our approach.

Key words: caching; XML; SOA; service-oriented architecture
History: Debabrata Dey, Senior Editor; Giri Kumar Tayi, Associate Editor. This paper was received on August 11, 2009, and was with the authors $10\frac{3}{4}$ months for 3 revisions. Published online in *Articles in Advance* September 15, 2011.

1. Introduction

As organizations strive to better utilize existing resources for competitive advantage, they often find that they need to integrate multiple business components (e.g., line-of-business applications) to automate business processes. In practice, these components are often heterogeneous along multiple dimensions—they reside on different platforms and are written in different programming languages. The principles of service-oriented architecture (SOA) promote the integration of such heterogeneous components through loosely coupled *services*.

Each SOA component service exposes a language-independent interface using a well-defined service protocol such as an XML service implemented using the SOAP or RESTful messaging protocols. (We note that binary-formatted messaging is possible though less frequently used in practice in SOAs, because it requires a tighter coupling between the service consumer and the service provider). Through this service interface, any client regardless of implementation language can access the service.

Organizations are increasingly deploying SOAs for a variety of business needs, e.g., integration with business partners. SOA growth is expected to continue—analysts at International Data Corporation (IDC, http://www.idc.com/about/about.jsp) predict a 7 × growth in SOA application development in the next five years (Rogers 2007).

Several benefits of SOA-driven applications are expected to propel this growth (Koch 2007). The most important benefit derives from the fact that service-oriented applications are composed of *reusable components*—unlike conventional applications that are composed of functions/classes created for use by a single application. In service-oriented systems, a component written once may be offered for reuse as a service by any application in need of that component's functionality. This, in turn, enhances many desirable properties of enterprise applications:

- Enterprise applications tend to be distributed in nature, which makes information consistency an essential property to support. Component reusability, i.e., the "code-once, use-many" characteristic of services, eliminates component redundancy and reduces inconsistency in underlying information.
- Because service components are *platform independent*, a service application written in Java running on a UNIX server may use the services of a component running on a Windows platform written in C#.
- Perhaps the greatest benefit of reusability is its ability to impact productivity and therefore cost in the software development life cycle. Virtually every step from development to testing to deployment is significantly impacted by the availability of reusable components.

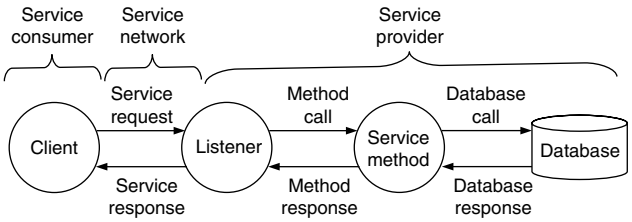
In spite of the many benefits of SOAs and associated systems, there are a few significant problems that need to be addressed. The two most important such issues are those of (a) *performance and scale*—perhaps the most well-known issues in service-oriented architectures (Dortch 2008), service-based systems (Liang et al. 2009), P2P service governance (Liang et al. 2006), and even cloud environments (Wu et al. 2009)—and (b) *security, privacy, and trust*. Though these latter issues are important (and deserve significant attention in their own right), our concern in this paper is the former, particularly in the context of service-oriented architectures. We begin with a discussion of the root causes and possible remedies for these concerns.

1.1. SOA Performance and Scale: Root Causes and Potential Remedies

To place this work in context, it is necessary to understand why service-based applications suffer from performance and scale issues. To see this, consider the anatomy of a service request/response interaction between a service consumer and a service provider.

Figure 1 shows a high-level view of such an interaction. Here, the service consumer sends a request, typically using an XML-based protocol for messaging, over the network to the service provider, where a listener receives the message and kicks off the appropriate service method. The service method performs the requested processing, perhaps accessing one or more databases in the course of processing. The service method then formats the result in a service message and returns it to the service consumer.

Figure 1 Service Process Example



In this context, let us consider the impact of the *distributed* and *reusable* properties of SOAs and their impact on SOA performance and scalability:

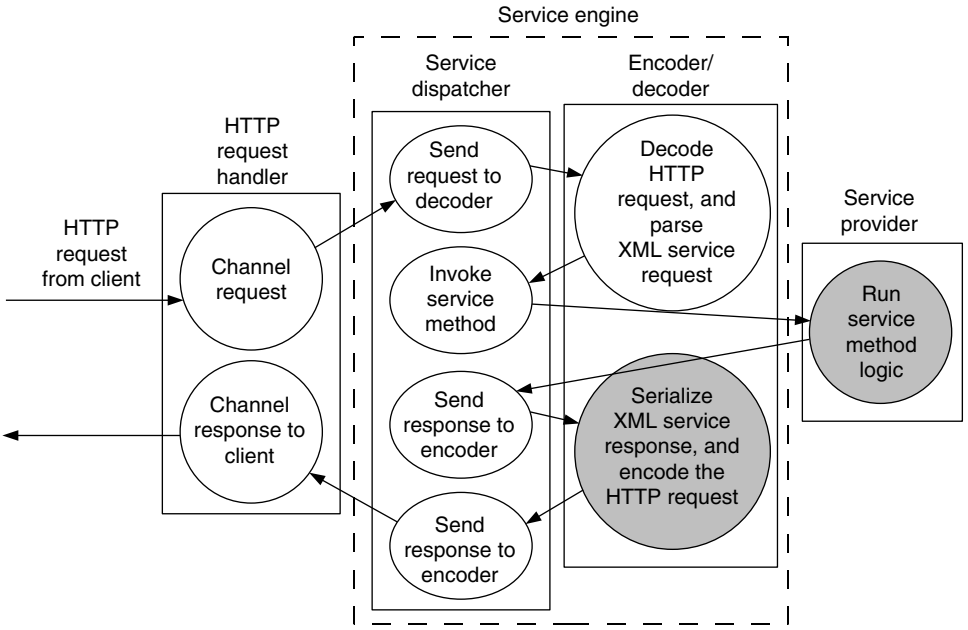
- Services in SOAs are designed to be reusable and to concentrate on the workload for a given service at a single location. In contrast, non-SOA implementations may replicate the same functionality within individual application silos.
- Because of the distributed nature of processing in SOAs, all requests and responses are passed across the network. Responses may contain large amounts of data that must travel from the service provider to the service consumer across the network.

First, we consider the processing-related consequences of concentrating the workload for a given function at a single location (and thus increasing the processing load there). In this context, let us drill down into the specific set of tasks that need to be accomplished by the service provider to respond to a service request.

Figure 2 shows the processing required of the service provider to respond to a service request in any XML-formatted service protocol. When a request arrives, the HTTP request handler separates the service request from the HTTP request and forwards it to the service engine, which deserializes the request, i.e., pulls the service method name and associated parameters from the XML-formatted request and invokes the application logic. The result of the application logic is an object, i.e., an in-memory representation of the result. The service engine takes this result object and serializes it into an XML-formatted message, i.e., it inserts the data from the result object into the XML format of the service response. When this is complete, it passes the message to the HTTP request handler, which wraps it into an HTTP message for transport and sends it out to the service consumer over the network.

Of all these steps, two are particularly compute-intensive (shown shaded in Figure 2), invariant of the messaging protocol used: running the application logic and serializing the result into an XML-formatted response message. These two steps account

Figure 2 A Deeper Look at Service Call Processing



for virtually all of the variation in server-side processing times across service requests. Intuitively, this makes sense. In terms of running the application logic, for example, one set of input parameters for a given service method might result in a very short processing time while another might require significantly more time. The time required to serialize a result object into an XML-formatted message varies directly with the amount of data to be inserted into the response (a study of the costs of serialization can be found in Hericko et al. 2003). The problem here is one of string processing—each data element in the response object must be copied into the XML format of the response. (This is similar to the object-relational impedance mismatch problem.)

Because these two steps account for the variation in processing time, they constitute the root cause of time-related performance problems in service message processing. One effective way to reduce this problem is to obviate these steps. It turns out that caching is a great way to do this, and it serves as the focus of this work.

Second, we consider the consequences of sending significant numbers of large messages across the network. XML messaging systems are responsible for significant and growing bandwidth usage on enterprise networks spread across geographically dispersed locations (Mimoso 2004). These consequences are obvious: As the number and size of messages increases, they can begin to clog the network, slowing the delivery of everything from email to network management alerts to ERP processing (and so on). Clearly, bandwidth consumption is a concern. In order to reduce bandwidth consumption, we need to send less data across the network, which we can achieve by storing cached content on the network closer to requestors.

1.2. Methodology and Contribution

In this paper, we examine the performance and scale problems in applications that were designed according to SOA principles using XML-based services. Having presented the nature and cause of these problems, we suggest a solution that utilizes the technique of *caching*. In particular, we demonstrate that caching, as a general technique, provides significant opportunities to mitigate large portions of delays inherent in these systems as well as to reduce network traffic caused by XML messaging. We then proceed to propose a specific technique to cache fragments of XML messages. This technique is made feasible by a recently introduced class of enterprise routing devices called XML routers. Finally, we analyze our solution using analytical as well as empirical modeling, and we demonstrate its effectiveness.

We follow a design science methodology in this effort. We draw on existing foundation work in caching and performance improvement as well as the current technology landscape in developing our method. We motivate our work by describing in detail the sources of performance issues in SOAs and by identifying gaps between existing solutions and the requirements of caching in the context of SOAs. We describe an implementable solution, including algorithms and a detailed architectural design. We evaluate our design through analysis and simulation.

We build an analytical model of the expected performance implications of our method and run a series of simulation and case study-based experiments to measure the expected and actual impacts of our method.

The main contributions of this work are as follows.

1. We identify the root causes of performance and scale issues in SOAs.

2. We introduce the notion of distributed fragment-oriented caching as a potential use case for XML router deployments to enhance SOA performance.

3. We specify an implementable scheme for service message fragment caching where cache storage locations are distributed on the XML router fabric.

4. We develop an analysis of the expected benefits of our scheme in terms of both bandwidth and response time impacts, and we describe a simulation study, the results of which show an up to 60% reduction in bandwidth consumption and up to 50% reduction in average response time.

5. We describe an implementation of our scheme integrated into commercial networking and middleware solutions.

6. We show the actual impact of our methods in a set of experiments in a staging environment. The results of these experiments demonstrate that when there is no resource bottleneck, the cache-enabled case reduces average response times by 40%–50% and increases throughput by 150% as compared to the no cache and fragment cache cases.

7. In a further experiment contrasting fragment caching and full message caching in the staging environment, we found that full message caching provides benefits when the number of possible unique responses is low (where the potential for reuse of a complete message is high), while the benefits of fragment caching increase as the number of possible unique responses increases.

8. We present a discussion of the practical decisions involved in deploying such schemes, e.g., situations where caching may not provide sufficient benefit to warrant its use as opposed to scenarios where caching can provide significant benefits.

The rest of the paper is organized as follows. In §2, we review the literature on Web content caching. In §3, we describe the component-oriented nature of service messages and the XML overlay networks that provide the framework for distributed caching. In §4, we present the intuition and system architecture for our caching scheme. In §5, we describe the design-time configuration and run time processing of our caching scheme. Section 6 presents a mathematical analysis of our scheme, §7 presents a set of simulation experiments, and §8 presents a set of experiments that show the actual impact of our methods in a staging environment where our methods are implemented on commercial networking and middleware hardware and software. In §9, we discuss the practical implications of our proposed scheme. We conclude the paper in §10.

2. Related Work

Using caching to improve the performance of Internet-accessible services is well-understood (Hosanagar et al. 2005, Mookerjee and Tan 2002). The idea behind caching is to provide improved delivery times, more scalable delivery networks, and lower loads on servers (Datta et al. 2003) by applying one or both of the following strategies: (a) *reuse computed results rather than recomputing them*, and (b) *serve cached items from a storage location as close to the requestor as possible to reduce network loads.* In this work, we consider the question of how to gain these benefits in an SOA scenario by applying caching techniques to fragments of service messages, which are typically implemented as XML-formatted messages.

We first consider whether existing Internet-based caching schemes can be directly applied to service messages. Virtually all such schemes are *proxy based*, i.e., cached content is stored in intermediary nodes that are situated between the content consumer and the content provider. Such caching schemes have been used to address both XML and HTML content. We consider each in turn.

Existing proxy-based content schemes also address HTML-centric content. Akamai's content delivery network (CDN)[1] is an example of a successful proxy-based network caching service. However, it is difficult, if not impossible to apply this technology to the service message caching case—the CDN proxy has no knowledge of the higher-level message of which its delivered content fragment is a component, and the recipient of the cached content is responsible for integrating the content into the larger message. This works because the content is virtually always cached at a granularity that the recipient (i.e., the browser) recognizes and can handle (i.e., full HTML pages or static image files). Consider the case where an HTML page refers to several image files stored in a CDN cache. Here, the CDN delivers the images from the cache to the user's browser, and the browser knows how to assemble and render the final page including all the images.

In the service message case, however, we cannot assume that the recipient is capable of reassembling a service message from a set of fragments. Individual fragments of service messages do not have meaning to the recipient without the context of the overall service message. Therefore, it is problematic (indeed,

[1] http://www.akamai.com/.

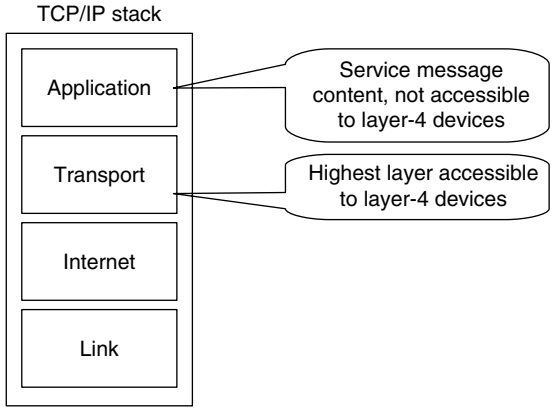
virtually infeasible) to transport fragments separately to the consumer—any fragment, when retrieved from a cache, must be included in the base message before the message is forwarded to the service consumer.

Further, proxy caching solutions have access only to the HTTP transport layer information of a service message, not the content within the XML message. Consider the TCP/IP network stack shown in Figure 3. Here, the XML content containing fragments of the service message that we wish to cache is part of the application layer. Proxy caches do not have access to this layer; rather, they can "see" only the HTTP transport layer wrapper around the service message. Thus, proxy caches cannot be used for service message content caching.

Datta et al. (2004) propose a proxy-based caching scheme for *fragments* of dynamically generated HTML, where the proxy is able to reassemble the cached fragments into a final message. This provides a way to cache fragments that a proxy can reassemble into full messages on a network. However, both this scheme and Akamai's CDN are designed to be deployed on the public Internet. In contrast, service message content delivery networks are mostly intraorganizational or limited to a set of business partners. Here, service messages traverse private networks, either behind firewalls or over VPNs, and carry corporate communication such as reports and internal transactions. RouteOne is an example of a consortium of companies communicating over a private network. This consortium consists of auto dealers and financial services organizations working together to process car loan applications (IBM 2006). Such organizations are loath to use public networks for such sensitive information, which makes public Internet proxies undesirable for use as caches.

Several commercial SOA caching solutions exist (Azim and Hamid 2002, Cohen 2006, Dattani et al. 2010, Powell 2002) for XML content. Though these solutions vary in their cache locations (e.g., at the consumer level, at the provider level, or as proxies on the network), they share a common trait: Virtually all of them cache full messages. Though such caching can be beneficial in some limited scenarios (e.g., when there is high demand for a small number of unique responses), its utility is limited in scenarios where the vast majority of requests require unique responses. We demonstrate this in §8. In many such scenarios, as we note above, fragments may be reusable across responses. In this work, we propose a method for caching such fragments to improve both response time and throughput performance for SOAs.

Figure 3 TCP/IP Stack Accessibility



Interestingly, an emergent infrastructural standard in network devices provides new possibilities for service message caching architectures in SOAs. Networks are increasingly being implemented on a new class of network fabric called XML overlay networks (XONs). XONs are composed of switches called XML routers that provide powerful XML processing and message routing functionalities, including protocol transformation, schema transformation, audit logging, and filtered routing based on message content.

Virtually every networking firm has major initiatives in this area (e.g., Cisco's application-oriented network (AON) module and IBM's DataPower products), and organizations are fast deploying these technologies to support their SOA initiatives (Research and Markets, Inc. 2003). For example, RouteOne uses IBM's DataPower appliances for wirespeed XSLT processing (a notorious CPU drain in applications) and to support encryption for individual XML elements. This implementation has significantly reduced processing times for loan applications while simultaneously providing increased security (IBM 2006).

Given the inadequacies of proxy-based schemes in addressing the caching needs for service message content and the emergence of the overlay network fabric implementations, we propose a new caching mechanism for *service message fragment caching in the XML overlay network*. This requires the coupling of caching logic and network routing, which is very different from caching schemes proposed to date. This is also different from the service message content caching proposals available in the literature today (e.g., Seltzsam et al. 2005, Tatemura et al. 2005, Tilkov 2009), which make assumptions that limit their applicability in this problem scenario.

Our fundamental observation is that the features offered by XML routers coupled with the fragment-oriented composition of service message content allow the development of fragment-oriented caching schemes, where the fragments can be stored in XON devices rather than in proxies. Two important characteristics of service messages and XONs make this idea feasible.

First, mechanisms exist to allow for fragment-oriented composition and the delivery of service message responses. In particular, the XPath construct allows for (1) viewing a service message as a composition of fragments, and (2) directly accessing specific service message components inside a message. This makes fragment-oriented caching feasible because it enables (a) the identification of a cacheable fragment within an XML response and (b) the identification of the location where a stored fragment should be inserted into a response.

Second, XML routers allow the specification of user-driven transformation and routing rules; this can exploit semantic knowledge about the application that is unavailable to the IP routers in the layer below. In other words, XML routers can "see" the application payload in the application layer, shown in Figure 3. This allows the XML router to be extended, for instance, with user-specified logic to process caching instructions in a service message.

Third, XON routers (e.g., Cisco's AON router) (Cisco Systems, Inc. 2008) typically contain a base caching infrastructure capable of storing and retrieving generic objects as well as maintaining distributed cache coherency across multiple XON instances. We will utilize this base caching infrastructure in our approach to handle basic cache management functionality. We build our fragment caching scheme on top of it.

XML caching has recently received much attention from researchers on aspects including caching full XML documents (Tilkov 2009), identifying reusable XQuery results based on previous query history (Chidlovskii and Borghoff 2000, Dar et al. 1996, Chen et al. 2002), and discovering frequent query patterns (Yang et al. 2003).

Tilkov (2009) discusses caching full XML documents at the HTTP layer (i.e., on network proxies). This caching method works well when full documents are reusable. However, in scenarios where documents are generated on the fly, full documents are rarely reusable as a whole document. Rather, fragments of the documents may overlap across response documents, making *fragments* a more useful granularity for caching. This is the case we consider in this work.

Tatemura et al. (2005) present a scalable middleware architecture for XML caching. The research question they address is how to keep the XML content in the cache updated in response to the dynamic changes made to back end XML data sources stored in relational databases. They define the cached XML data as views of the relationally stored data residing in the back end relational database, and they provide declarative ways to access these XML data. An XML-specific view maintenance technique can then be applied to discover discrepancies between the cached content and origin data. Though this work does discuss XML data cached at the provider, the focus of Tatemura et al. (2005) is on view maintenance with the back end data source rather than on supporting efficient distributed caching in for message content, which is our purpose here.

Seltzsam et al. (2005) study ways to present semantic caching. They develop an XML-based declarative language to annotate WSDL (Web Service Description Language) documents with cache-relevant semantics to allow for proxylike caching of XML fragments on the network. Seltzsam et al. (2005) propose a mapping mechanism to determine whether or not the full set of content needed to respond to a request is present in the cache. This work differs from ours along several dimensions. First, in their proposed scheme, each cache instance manages cache eviction and invalidation independently, which allows for inconsistency across cache instances. In contrast, our method proposes a distributed cache, which ensures consistency across all cache instances. Second, their proposed scheme addresses only situations where all content in a response is cacheable—it cannot handle request types where responses contain a mixture of cacheable and noncacheable content. Third and finally, their proposed method will serve cached content from network locations only when all needed content is available in the local cache; in all other situations, the provider services the request. In contrast, our work addresses the more general case of mixing cacheable and noncacheable content in responses as well as the case where only part of the required content resides in network caches.

Olston et al. (2005) describe a secure and plug-in (provided by third party) scalability service for applications. It is achieved by caching home organizations' code and data in a distributed architecture of proxy servers, each of which has a duplicate Web server, application server, and database server. The focus of the research is distributed consistency management for query and update operations in the database servers of the proxy servers. Our work shares with the research of Olston et al. (2005) the problem space requirements of caching dynamic content and of not storing sensitive data outside the organization. However, our caching scheme is not proxy server-based. The caching nodes in our scheme have the capability to both read and understand caching instructions embedded in service message content as well as retrieve and store fragments according to those instructions.

Lelli et al. (2006) propose a caching-based scheme to address XML parsing delays by caching parsing processing results of previously parsed XML documents, thereby speeding up the XML parsing process. This work is complementary to our own; it is possible to use both schemes simultaneously.

3. Background

To understand our work, it is necessary to gain an appreciation of two background areas—component-oriented service message documents and XML overlay networks.

3.1. Component-Oriented Service Message Documents

In this section, we provide an overview of service message documents with a special emphasis on explaining how the component-oriented nature of such documents gives rise to the cacheability semantics that we will exploit in this paper.

We present an example scenario to illustrate the structure of service message request/response interaction. Subsequently, we use this scenario as a running example throughout the remainder of the paper. Our example scenario considers the case of a large construction materials supplier with distributed operations and a distributed information system. Separate, loosely coupled information systems serve different functional needs—`Finance/Accounting`, `Sales`, `Human Resources`, and `Shipping`. Functional units, particularly `Shipping`, are widely dispersed because different shipping locations service different geographic regions.

In this example scenario, we consider the specific case of purchase orders (POs), which are generated within the `Sales` system but are used throughout the organization. `Accounting` uses POs to generate invoices, `Finance` accounts for the transactions on the books, `Human Resources` pays any applicable sales commissions, and `Shipping` ensures that requested items are delivered to customers.

Within the `Sales` system, a PO is represented as a class of objects, depicted in the diagram shown in Figure 4. A PO is an aggregation of several object types: a `Customer`, i.e., the party actually paying for the construction; a `Contractor`, i.e., the party placing the order for materials needed; an `OrderHeader`, which summarizes the order including order totals; and a set of `Lines`, each of which specifies a `Product` and the quantity requested and any special handling requirements.

The `Sales` system makes PO data available through a service. Table 1 shows an example request for PO data while Table 2 shows the corresponding response. In the request, the requestor provides an `OrderNumber` (here, 00099999) and expects the associated PO data to be returned in the response. Our example here is presented in generically formatted XML for the purposes of illustration, but our methods will work for any XML-based messaging protocol.

The response, shown in Table 2, contains an XML-formatted representation of the `PurchaseOrder` object shown in Figure 4, with well-defined sections of the response mapping directly to the "parts" of the `PurchaseOrder` class of the same names, e.g., there is a `Customer` section in the response that contains data values for the attributes of the `Customer` portion of the `PurchaseOrder` class diagram.

Figure 4 Class Diagram of the Purchase Order Example

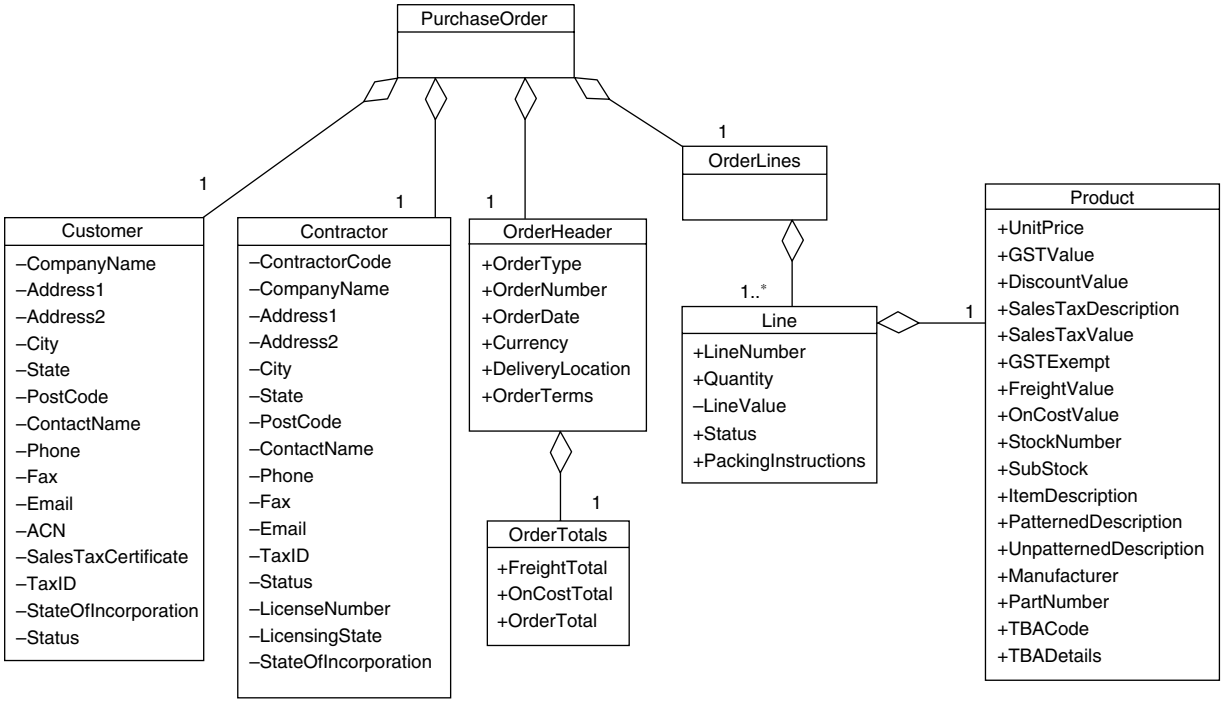


Table 1 Example XML Service Request

```
<Request
xmlns:p="[providerNamespace]">
  <p:getPurchaseOrder>
    <p:OrderNumber>00099999</p:OrderNumber>
  </p:getPurchaseOrder>
</Request>
```

We now consider the response in Table 2 more closely. A given `Contractor` may be involved in multiple projects at a given time—perhaps hundreds for a large construction company. A given `Customer` may be involved in multiple construction projects (not necessarily with the same contractor). A given `Product` may be ordered through any `PurchaseOrder`. Clearly, the contents of these portions of the response messages are reusable across multiple responses, where those responses require the same data for `Customer`, `Contractor`, or `Product`. Only the content pertaining to the `OrderHeader` and the non-product-related data (e.g., quantity ordered) in the `Lines` is specific to a single `PurchaseOrder` and cannot be reused. Figure 5 shows a simple graphical depiction of the example `PurchaseOrder` response message, where cacheable fragments are shown as shaded squares and noncacheable fragments are shown as unshaded squares.

It turns out that much of the service-oriented content exchanged over intraorganizational service networks can be broken down into cacheable and noncacheable fragments, where much of the content is reusable across multiple responses.

3.2. XML Overlay Networks

It is well-known that the vast majority of commercially deployed service applications are inside organizations (as opposed to on the public Internet). Indeed, in many large organizations (Verton 2003, Gruman 2005), data and document exchange is almost all XML centric. This has given rise to a new class of networking fabric inside organizations called XML overlay networks (XONs). These networks sit on top of regular IP networks and provide value-added functionality such as XML processing, plug-and-play semantic routing, cross-application data integration, and a host of other functions including user-defined functionality. The heart of the XON is the XML router. The ability of XML routers to process service message content (and act as a cache) is one of the major drivers of this work. Thus, it would be helpful to provide the reader with a high-level overview of XML routers.

An XML router is a layer-7 switch, meaning it has access to data from the service message itself that other network devices do not have (layer-4 switches have access only to the transport wrappers that encode destination and other transport-related information). This gives the XML router access to the full content of a service message that is passed over the network. Most commercially available XML routers provide the ability for developers to build custom logic to run on the routers using standard well-known programming languages that enable the XML caching functionality reported in this paper.

An XML router can provide several types of services on the network (Anthias and Sankar 2006), including XML processing (e.g., XPath routing or XSL transformation, etc.), security processing (e.g., encryption/decryption), message transformation (e.g., converting between different B2B interchange schemes), and load balancing. Enabling such processing on an XML router has two benefits: (1) it reduces overhead on servers that do application processing by offloading high-CPU tasks; and (2) it allows the XML router to make policy-based decisions based on user-defined policies without having to consult a line-of-business application.

An XML router is meant to complement the IP routers and switches that carry streams of data traffic across the Internet. It does not interfere with or replace TCP/IP protocols but rather adds new functionality to a network. Consider Figure 6 (based on Fenner et al. 2005), which shows a set of XML routers deployed alongside IP switches in a TCP/IP network. Whereas the IP switches can access only transport layer information, e.g., where a message is going and how it should be treated along the way, XML routers have the ability to access application layer content, i.e., they can access service message content and act upon it (see Figure 3). These XML routers form a separate logical layer of devices within a network, and this "layer" adds the ability to execute logic on the network.

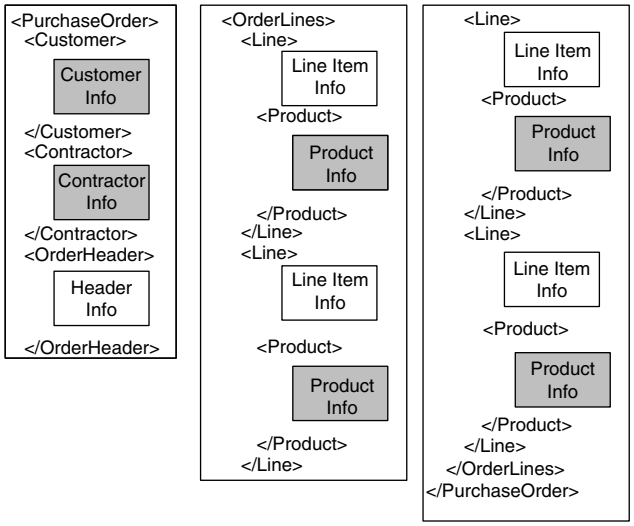
Here, a message from provider P_1 arrives at an XML router XR_1 from a TCP/IP switch IP_1. The XML router examines the message information against the policies and services configured on it to determine if the message is of interest. Recognition of messages of interest can be based on any observable message attribute including its URL, an encoding/encryption scheme, or even a specific query in XQuery. For example, policies can be defined by path expressions and user-defined functions using XQuery.

If XR_1 finds that the message matches one of its configured policies, the processing specified within the policy executes (perhaps XSLT processing, a compression scheme transformation, or modified routing instructions), after which the message is sent toward its destination through the TCP/IP network. If the message is not of interest to the XML router, it continues on its way unimpeded and unchanged via the TCP/IP network. This trip might be routed through one or more other XML routers in the network, e.g., perhaps to log service access by a given consumer application for auditing or compliance purposes.

Table 2 Example XML Service Response

```
<Response
xmlns:p = "[providerNamespace]">
  <p:PurchaseOrder>
    <p:Customer>
      <p:CompanyName>OCS Fictional Company<CompanyName>
      <p:Address1>ADDRESS LINE 1</Address1>
      <p:Address2/>
      <p:City>MELBOURNE</City>
      <p:State>VIC</State>
      <p:PostCode>3182</PostCode>
      <p:ContactName>Jim Smith</ContactName>
      <p:Phone>03 99999999</Phone>
      <p:Fax>03 99999999</Fax>
      <p:Email>jim.smith@ocs-fict.co.au</Email>
      <p:ACN>99 999 999</ACN>
      <p:SalesTaxCertificate>E9999999</p:SalesTaxCertificate>
      <p:TaxID>123-89-4567</p:TaxID>
      <p:StateOfIncorporation>VIC</p:StateOfIncorporation>
      <p:Status>Active</p:Status>
    </p:Customer>
    <p:Contractor>
      <p:ContractorCode>9999</p:ContractorCode>
      <p:CompanyName>SUPPLIER XYZ</p:CompanyName>
      <p:Address1>ADDRESS LINE 1</p:Address1>
      <p:Address2/>
      <p:City>MELBOURNE</p:City>
      <p:State>VIC</p:State>
      <p:PostCode>3182</p:PostCode>
      <p:ContactName>John Bloggs</p:ContactName>
      <p:Phone>03 88888888</Phone>
      <p:Fax>03 88888888</Fax>
      <p:Email>john.bloggs@xyz-supp.co.au</Email>
      <p:ACN>88 888 999</ACN>
      <p:SalesTaxCertificate>E8888888</p:SalesTaxCertificate>
      <p:TaxID>456-12-7893</p:TaxID>
      <p:StateOfIncorporation>VIC</p:StateOfIncorporation>
      <p:Status>Active</p:Status>
    </p:Contractor>
    <p:OrderHeader>
      <p:OrderType>ST</p:OrderType>
      <p:OrderNumber>00099999</p:OrderNumber>
      <p:Currency>AU</Currency>
      <p:DeliveryLocation>
        OCS CENTRAL WAREHOUSE ADDRESS
      </p:DeliveryLocation>
      <p:OrderTerms>NET FROM STATEMENT 30 DAYS</p:OrderTerms>
      <p:OrderTotals>
        <p:FreightTotal>0.00</p:FreightTotal>
        <p:OnCostTotal>0.00</p:OnCostTotal>
        <p:OrderTotal>702.08</p:OrderTotal>
      </p:OrderTotals>
    </p:OrderHeader>
    <p:OrderLines>
      <p:Line>
        <p:LineNumber>0001</p:LineNumber>
        <p:Quantity>1.0000</p:Quantity>
        <p:LineValue>21.00</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
        <p:Product>
          <p:UnitPrice>21.0000</p:UnitPrice>
          <p::GSTValue>2.10</p:GSTValue>
          <p:DiscountValue>0.00</p:DiscountValue>
          <p:SalesTaxDescription>TAX EXEMPT</p:SalesTaxDescription>
          <p:SalesTaxValue>0.00</p:SalesTaxValue>
          <p:GSTExempt>N</p:GSTExempt>
          <p:FreightValue>0.00</p:FreightValue>
          <p:OnCostValue>0.00</p:OnCostValue>
          <p:StockNumber>00004004</p:StockNumber>
          <p:SubStock>01</p:SubStock>
          <p:ItemDescription>
            FLANGE PIPE BLACK 250MM NB
            BST D 10IN ABC
          </p:ItemDescription>
          <p:PatternedDescription/>
          <p:UnpatternedDescription/>
          <p:Manufacturer>GENERAL MFG</p:Manufacturer>
          <p:PartNumber>03230505</PartNumber>
          <p:TBACode/>
          <p:TBADetails/>
        </p:Product>
      </p:Line>
      <p:Line>
        <p:LineNumber>0002</p:LineNumber>
        <p:Quantity>50.0000</p:Quantity>
        <p:LineValue>18.50</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
        <p:Product>
          <p:UnitPrice>0.3700</p:UnitPrice>
          <p:GSTValue>0.037</p:GSTValue>
          <p:DiscountValue>0.00</p:DiscountValue>
          <p:SalesTaxDescription>TAX EXEMPT</p:SalesTaxDescription>
          <p:SalesTaxValue>0.00</p:SalesTaxValue>
          <p:GSTExempt>N</p:GSTExempt>
          <p:FreightValue>0.00</p:FreightValue>
          <p:OnCostValue>0.00</p:OnCostValue>
          <p:StockNumber>00006219</p:StockNumber>
          <p:SubStock>01</p:SubStock>
          <p:ItemDescription>
            TAPE THREAD SEALING 12MM X 10M TEFLON PTFE
          </p:ItemDescription>
          <p:PatternedDescription/>
          <p:UnpatternedDescription/>
          <p:Manufacturer>BLKWOODS</p:Manufacturer>
          <p:PartNumber>05122404</p:PartNumber>
          <p:TBACode/>
          <p:TBADetails/>
        </p:Product>
      </p:Line>
      <p:Line>
        <p:LineNumber>0003</p:LineNumber>
        <p:Quantity>30.0000</p:Quantity>
        <p:LineValue>457.50</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
        <p:Product>
          <p:UnitPrice>15.2500</p:UnitPrice>
          <p:GSTValue>1.525</p:GSTValue>
          <p:DiscountValue>0.00</p:DiscountValue>
          <p:SalesTaxDescription>TAX EXEMPT</p:SalesTaxDescription>
          <p:SalesTaxValue>0.00</p:SalesTaxValue>
          <p:GSTExempt>N</p:GSTExempt>
          <p:FreightValue>0.00</p:FreightValue>
          <p:OnCostValue>0.00</p:OnCostValue>
          <p:StockNumber>00004710</p:StockNumber>
          <p:SubStock>01</p:SubStock>
          <p:ItemDescription>
            2-INCH PVC BALL VALVE
          </p:ItemDescription>
          <p:PatternedDescription/>
          <p:UnpatternedDescription/>
          <p:Manufacturer>HEARTLAND</p:Manufacturer>
          <p:PartNumber>02732618</p:PartNumber>
          <p:TBACode/>
          <p:TBADetails/>
        </p:Product>
      </p:Line>
      <p:Line>
        <p:LineNumber>0004</p:LineNumber>
        <p:Quantity>2.0000</p:Quantity>
        <p:LineValue>205.08</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
        <p:Product>
          <p:UnitPrice>102.5400</p:UnitPrice>
          <p:GSTValue>10.025</p:GSTValue>
          <p:DiscountValue>0.00</p:DiscountValue>
          <p:SalesTaxDescription>TAX EXEMPT</p:SalesTaxDescription>
          <p:SalesTaxValue>0.00</p:SalesTaxValue>
          <p:GSTExempt>N</p:GSTExempt>
          <p:FreightValue>0.00</p:FreightValue>
          <p:OnCostValue>0.00</p:OnCostValue>
          <p:StockNumber>00006219</p:StockNumber>
          <p:SubStock>01</p:SubStock>
          <p:ItemDescription>
            3/4-INCH IPS Union x 3/4-INCH Sweat ISOLATION VALVE
          </p:ItemDescription>
          <p:PatternedDescription/>
          <p:UnpatternedDescription/>
          <p:Manufacturer>JOHNSON PLUMBING</p:Manufacturer>
          <p:PartNumber>16384472</p:PartNumber>
          <p:TBACode/>
          <p:TBADetails/>
        </p:Product>
      </p:Line>
    </p:OrderLines>
  </p:PurchaseOrder>
</Response>
```

Figure 5 Cacheable Fragments of a Service Response



4. Intuition and Caching Architecture

The intuition behind our proposed scheme is simple and consists of four related activities: (a) generating a cache-enabled response; (b) recognizing and caching the cacheable fragments of a service message response in the routers of the response path; (c) serving cached content from the network; (d) determining when cached fragments can be used in a new response; and (e) computing a best-cost routing for a new response through the overlay network from provider to consumer.

Our method departs from traditional caching schemes in two significant ways: (1) caching occurs in the network nodes (as opposed to specialized proxies), and (2) on-demand routing is performed. Typically, caching schemes are independent of routing schemes; however, they are closely coupled in our case. To make things clearer, we now provide a high-level overview of three important concepts for our approach: (a) how caching logic is architecturally deployed on the network; (b) how caching-related information is passed from service providers to cache-enabled XON routers; (c) how cache-enabled routers process caching-related information. We elaborate on the details of these three concepts in later sections of this paper.

We begin with a discussion of the caching architecture on a network. Figure 7 depicts this architecture. Service message requests and responses are exchanged between applications (App_1, App_2, App_3, App_4, and App_5) that are geographically distributed. These applications communicate among themselves using service messages. The messages are routed through the common TCP/IP network overlaid with XML routers (shown in Figure 7 as lightly shaded circles) as described in §3. Both the service provider applications as well as the XON routers are equipped with caching logic, shown in Figure 7 as a cache directory for service providers and as caches (depicted as darker-shaded circles) colocated with XON routers.

Service providers pass caching information to a cache-enabled router by inserting caching instructions into service messages. These instructions tell the routers to either *get* (i.e., retrieve) cached fragments from a local cache and insert them into the service message or copy specified fragments from the service message and *set* (i.e., store) them to the local cache. For example, consider the service response depicted in Figure 5, which shows cacheable fragments as shaded boxes. Figure 8 depicts this same response, with caching instructions included, for the case where the XON router that will process the caching instructions has the needed `Customer` fragment and all four `Product` fragments stored in its cache but will need to store the `Contractor` fragment to cache. In this case, the service provider generates a service response message that omits the cacheable fragments and inserts a

Figure 6 XML Overlay Network Architecture

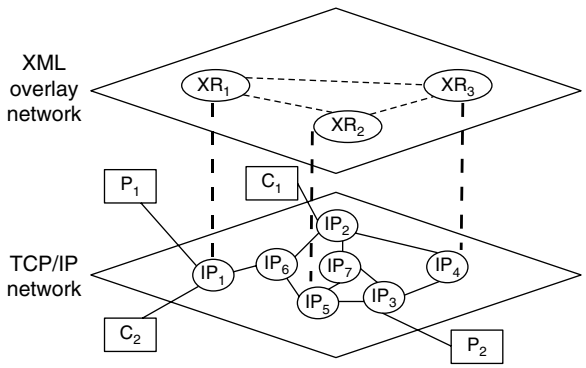


Figure 7 XML Network Caching Architecture

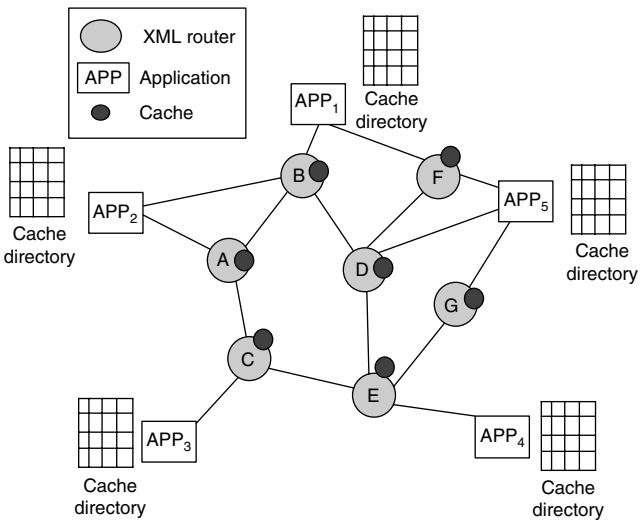
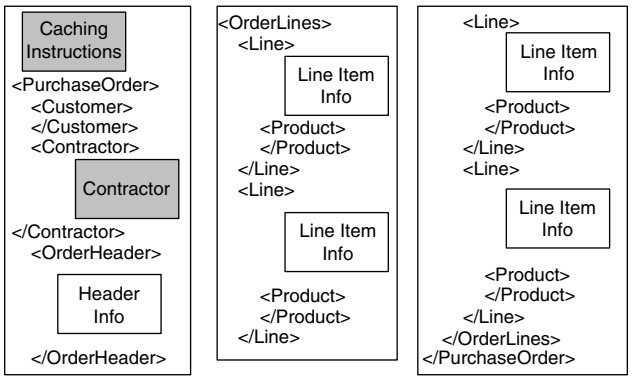


Figure 8 Cache-Enabled Service Response



set of caching instructions that describe which fragments need to be drawn from the XON's cache and which fragments need to be stored to cache.

Figure 9 depicts the run time processing for service messages when network-based caching is in place, using the PO scenario to illustrate the caching process. When a service consumer sends a request to the service provider (step 1), the provider runs the associated code to generate the response (step 2), which includes inserting any necessary caching instructions into the response. If the response requires cache processing on the network, the service provider selects an XON router to perform the cache processing (step 3) based on a best-cost calculation, which takes into consideration the contents of an XON router's cache and its location on the network. The service provider then sends the response to the selected XON router (step 4). When an XON router receives a response message with caching instructions, it performs the specified operations (i.e., getting and setting cache fragments) to transform a cache-enabled response message (e.g., as seen in Figure 8) into a complete service message (e.g., as seen in Figure 5), identical to the one that would have been generated by the service provider if caching were not in place (step 5). When all cache processing is complete, the XON router sends the response message on to the requestor (step 6).

Having described how our approach works at a high level, we now discuss the details of how our methods work.

5. Details of Our Method

Our proposed mechanism impacts an XML overlay network to act as a caching fabric for service message document exchanges. More specifically, our methods impact the service message response process in four ways: (a) we add logic to the service provider to enable local caching; (b) we add certain control information to service message response documents to make fragments of these documents cache ready; (c) we propose a semantic routing mechanism to select a delivery path for each service response from the service provider to the service consumer; and (d) we add logic to the XON routers in the XML overlay network to enable the routers to reconstitute service messages using locally cached fragments.

We first describe the architectural components we add to the service provider and to the XON routers to add caching logic, and then we go on to describe design time configuration details to enable caching within service applications. Finally, we describe run time processing at the service provider and XON routers to provide cache-enabled services.

5.1. Architectural Details

Figures 10 and 11 depict the cache-enabling architectural components (shown shaded) on the service provider and an XON router, respectively. In this section, we describe each in turn.

Processing on the service provider in a scenario without caching implemented proceeds as follows. When a service request arrives at the provider, an

Figure 9 Run Time Processing Example

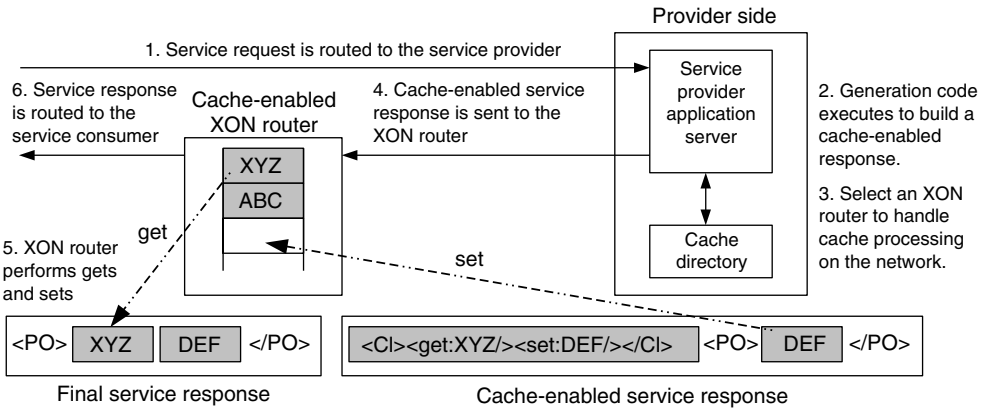
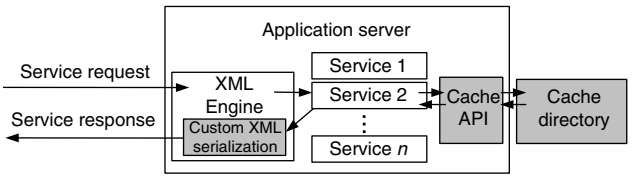


Figure 10 Cache-Enabling Components on the Service Provider



XML engine unpacks the XML-formatted request information, determines which service has been requested, and invokes the appropriate service with any input data included in the request. The service performs the requested processing and produces a programmatic object representing the response. For example, in the PO example, the provider would generate an instance of the `PurchaseOrder` class (see Figure 4) populated with the appropriate values for the PO number in the request. The XML engine then performs a process called *XML Serialization* to transform the `PurchaseOrder` object into an XML-formatted `<PurchaseOrder>` (as in Table 2) that is suitable for inclusion in a service response.

In a caching scenario, three components enable network-based caching on the service provider: (a) a cache directory; (b) a cache application programming interface (API); and (c) a custom XML serialization library. The cache directory allows the provider to store and retrieve programmatic objects from applications through a cache API. This allows a service to check the cache directory for needed objects before generating them, and to avoid the processing cost of generating objects available from the cache (locally or on the network). The custom XML serialization library inserts any needed caching instructions (i.e., `get` and `set` calls) so that a service message can be reconstituted with network-based cached fragments. The library also decides which router should handle cache processing on the network.

For each object cached on the service provider, the cache maintains a number of data elements to support caching. These data elements are described in Table 3. Most of these elements are required to support generic caching functions—any caching system will require a unique identifier for each stored element, a time-to-live value, a size for the element,

Figure 11 Cache-Enabling Components on the XON Router

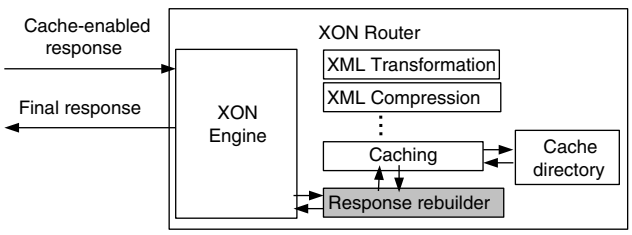


Table 3 Provider Cache Metadata

Provider cache element	Description
ElementID	Unique identifier for the stored object
ValidList	Which nodes in the XML network have the object's XML fragment stored in cache
Time to live	Timestamp indicating the end of the fragment's validity period
Size	Number of bytes in the stored element
Content	Content of the element

and the contents of the element itself. To support caching on the network, we maintain an additional piece of information for each stored element: a list of XON routers where the cached element is currently cached locally that supports the decision of which router will process the cache-enabled response on the network. Thus, across all providers in the enterprise, each application's cache directory has knowledge of which routers contain its fragments but no knowledge of where other providers' fragments might be cached.

On the XON router, we take advantage of existing generic caching functionality already available on the router, e.g., as provided by the Cisco AON router (Cisco Systems, Inc. 2008), along with standard XML processing functionality such as XSLT, XML compression, schema validation, logging, and other functions. We extend this standard functionality with a custom plug-in (labeled "response rebuilder" in Figure 11) that interprets the caching instructions in a cache-enabled response, performs the required `get` and `set` operations, and reconstitutes the final service response for delivery to the service consumer. (We keep our discussion generic here, and then describe the details of our integration with a commercial XON router in §8.)

In the cache directory on the XON router, we maintain the same generic caching data as the service provider but, rather than keeping a list of all routers where the fragment is valid, we store the identity of the service provider that generated the fragment (as shown in Table 4). This identity is used in cache maintenance processing (as we will describe

Table 4 Router Cache Metadata

Router cache element	Description
FragmentID	Unique identifier for the fragment
ProviderID	Unique identifier for the application providing the fragment
Time to live	Timestamp indicating the end of the fragment's validity period
Size	Number of bytes in the fragment
FragContent	Content of the fragment

in §5.3). Thus, each router in the enterprise XML overlay network knows which fragments it stores for each provider application but not which fragments other routers may have in their cache instances.

Having described the architectural elements of our approach, we now describe cache processing, including design time configuration requirements.

5.2. Run Time Algorithms

At run time, when a service message request arrives at a provider application, our proposed mechanism attempts to ensure low-cost delivery of the response by using both caching and semantic routing methods. Within this architecture, the following work-flow ensues:

1. The provider generates a cache-enabled response. This means that the provider generates a response document where any potential cacheable fragments are either (a) already inserted in the document and ready to be cached en route to the service consumer, or (b) marked as `get` ready, to be filled in by nodes in the intervening network. Any needed caching instructions are included.

2. Having generated the cache-enabled response document, the provider then decides how to route this response to the requester. At this stage, this involves identifying a coordinating router (CR), a specific node in the XON fabric to be responsible for cache processing on the network and routing the cache-enabled response there.

3. The CR receives the cache-enabled response from the provider and inserts all fragments needed to complete the response. It might do so by finding such fragments in its own cache or by collaborating with other neighboring nodes that might have needed fragments residing in cache.

4. Finally, the CR sends the complete message to the final recipient.

We next describe these steps as well as the details of cache maintenance in our approach.

5.2.1. Generating a Cache-Enabled Response. Effectively, we need to impact a service message document in the following ways to make it cache-ready: (a) we need to mark objects in the provider application as cacheable so that the service provider can avoid the processing cost of generating them when they are available from cache; and (b) we need to generate caching instructions in the service message so that the cache-enabled router can perform the necessary processing to reconstitute the final service message.

We mark a class of objects as cacheable by adding an `isCached` attribute to the class as shown in Table 5. This attribute plays a dual role. First, its presence indicates to other classes in the provider application that objects of the class are cacheable (the presence or absence of an attribute can be checked via reflection mechanisms in object-oriented frameworks). Second, its value (`true` or `false`) indicates whether the fragment generated from the object is available on an XON router in the network.

Table 5 Marking a Class of Objects as Cacheable

```
class Customer
{
  // member variables
  String name;
  ...
  ...
  ...
  transient int isCached = false;
  String getPrimaryKey( );
}
```

We instrument the class methods to check for an object's presence in cache. Algorithm 1 shows a pseudocode representation of such logic in the context of our `getPurchaseOrder` example. Here, for each cacheable object, e.g., for the `Customer` object, the service first checks whether the needed object is available in cache using the `getCache` call. If the object is found in cache and exists on the network (i.e., the `ValidList` returned from cache is not empty), the service retrieves the object from cache (line 8), sets the `isCached` attribute value to `true` (line 9), and *skips the processing required to retrieve the object's attribute values from the database*. If the object does not exist in the provider's cache (and is therefore not available on the network), the service populates the object's attributes with a database call (line 11), sets the `isCached` attribute to `false` (meaning that the associated XML fragment should be stored to caches on the network (line 12)), and stores a serialized form of the object to the provider's local cache (line 13). This type of application instrumentation is standard practice in caching applications (see, e.g., examples from Oracle Corporation 2010, Limaye 2005, and open-source caches Apache Jakarta Java Caching System Project 2010 and OpenSymphony OSCache Project 2010).

Algorithm 1 (Checking for an object in cache)

```
1: <WebService>
2: PurchaseOrder getPurchaseOrder
       (String orderNumber)
3: {
4: PurchaseOrder p=new PurchaseOrder( );
5: // get customer
6: if checkCache("Customer:CompanyName
   =OCS Fictional Company") exists then
7:     p.Customer=getCache
         ("Customer:CompanyName
         =OCS Fictional Company");
8: p.Customer.isCached=true;
```

```
 9: else
10:    p.Customer=Select*from Customer where
         CompanyName='OCS Fictional
           Company';
11:    p.Customer.isCached=false;
12:    p.Customer.setToLocalCache( );
13: //get contractor
14: if checkCache('Contractor:CompanyName
    =SUPPLIER XYZ') exists then
15:    p.Contractor=getCache
         ('Contractor:CompanyName
         =SUPPLIER XYZ');
16:    p.Contractor.isCached=true;
17: else
18:    p.Contractor=Select*from Contractor
        where CompanyName='SUPPLIER XYZ';
19:    p.Contractor.isCached=false;
20:    p.Contractor.setToLocalCache( );
21: // get OrderHeader
22: ...
23: // get OrderLines
24: ...
25: return p;
26: }
```

When the cache-enabled XML serialization library finds a cache-enabled object while converting an object from its programmatic form to XML, it inserts instructions for extracting cacheable content and storing it for reuse (in caching terms, a `set` instruction) or inserting already cached content inside a document (i.e., a `get` instruction). We first describe the details of these caching instructions and then describe how the serialization library actually inserts them into response messages at run time.

We describe these instructions using the service message document instance shown in Table 6. This document is an example of a cache-ready response and is based on our example service response shown in Table 2.

The `CachingInstructions` element contains caching information that the XML router will use in cache processing for the message. Within `CachingInstructions`, the `set` element defines a list of `Fragment` elements to be stored in cache on the network. Each such `Fragment` element provides caching metadata for a particular object `type`, e.g., a `Customer` or `Product` object, with a unique `fragmentID` and a `ttl` value. The fragments to be set to cache reside in the `PurchaseOrder` portion of the message, at a location denoted by the XPath expression in the `path` element in the set instruction.

Within `CachingInstructions`, the `get` element defines a list of `Fragment` elements to be retrieved from cache on the network and inserted into the response message. The instructions to retrieve a fragment from cache contain a `type` attribute that maps to the object type the fragment represents, a `ValidList` that lists the routers where the element is available in cache, and an XPath expression in a `path` element that points to the location in the `PurchaseOrder` where the fragment should be inserted. Note that the cache-enabled response message contains no reference within the `PurchaseOrder` element to fragments that need to be retrieved from cache until the XON router fills them in during cache processing.

This completes our description of the `CachingInstructions` information. We next describe how the XML serialization library inserts caching instructions at run time.

Algorithm 2 describes the logic that the XML serialization library uses to insert caching instructions into the response message. For each object o to be serialized into the response, the library checks for the existence of an `isCached` attribute. If o has an `isCached` attribute and its value is `true`, then the library generates the appropriate insertion XPath expression (line 4), retrieves the `ValidList` of o from cache (line 5), and inserts a `get` instruction into the `CachingInstructions` (line 6). If o has an `isCached` attribute and its value is `false`, then the XML fragment of o needs to be set to cache on the network. The library serializes o and inserts the resulting XML into the response message (line 8), generates the appropriate location XPath expression to find the XML fragment of o at `set` time (line 9), retrieves the `ttl` value of o from cache (line 10), and inserts the appropriate `set` instruction into the `CachingInstructions` (line 11). If o has no `isCached` attribute, then o is not cacheable and the library simply serializes o and inserts the resulting XML into the response message (line 13).

Algorithm 2 (Inserting caching instructions during XML serialization)

```
 1: for all object o to be serialized into
      the response message do
 2:    if o.hasAttribute("isCached") then
 3:       if o.isCached==true then
 4:          path=generateInsertionXpath( )
 5:          ValidList = o.getValidList( )
 6:          insert get instruction for o, with path
               and ValidList
 7:       else
 8:          serialize o and insert into response XML
 9:          path=generateLocationXpath( )
10:          ttl=o.getTtl( )
11:          insert set instruction for o, with path
               and ttl
12:    else
13:       serialize o and insert into response XML
```

Having described the structure of our caching instructions as well as run time cache processing at

Table 6 Example Cache-Enabled XML Service Response

```
<Response
xmlns:p = "[providerNamespace]
xmlns:c = "[enterpriseCachingNamespace]">
  <c:CachingInstructions>
    <c:Set>
      <c:Fragment type = "Customer"
        fragmentID = "Customer:CompanyName = OCS Fictional Company">
        <c:path>
          /p:PurchaseOrder/p:Customer
          [@p:CompanyName = "OCS Fictional Company"]
        </c:path>
        <c:ttl unit = "minutes">360<c:ttl>
      </c:Fragment>
    < /c:Set>
    <c:Get>
      <c:Fragment type = "Contractor"
        fragmentID = "Contractor:CompanyName = SUPPLIER XYZ">
        <c:path>
          /p:PurchaseOrder
        </c:path>
        <c:ValidList>
          <c:Node>200.17.16.17</c:Node>
        <c:ValidList>
      </c:Fragment>
      <c:Fragment type = "Product"
        fragmentID = "Product:StockNumber = 00004004">
        <c:path>
          /p:PurchaseOrder/p:OrderLines/p:Line[@p:LineNumber = "0001"]
        </c:path>
        <c:ValidList>
          <c:Node>200.17.16.21</c:Node>
        <c:ValidList>
      </c:Fragment>
      <c:Fragment type = "Product"
        fragmentID = "Product:StockNumber = 00006219">
        <c:path>
          /p:PurchaseOrder/p:OrderLines/p:Line[@p:LineNumber = "0002"]
        </c:path>
        <c:ValidList>
          <c:Node>200.17.16.14</c:Node>
          <c:Node>200.17.16.21</c:Node>
        <c:ValidList>
      </c:Fragment>
    < /c:Get>
  < /c:CachingInstructions>
  <p:PurchaseOrder>
    <p:Customer>
      <p:CompanyName>OCS Fictional Company<CompanyName>
      <p:Address1>ADDRESS LINE 1</Address1>
      <p:Address2/>
      <p:City>MELBOURNE</City>
      <p:State>VIC</State>
      <p:PostCode>3182</PostCode>
      <p:ContactName>Jim Smith</ContactName>
      <p:Phone>03 99999999</Phone>
      <p:Fax>03 99999999</Fax>
      <p:Email>jim.smith@ocs-fict.co.au</Email>
      <p:ACN>99 999 999</ACN>
      <p:SalesTaxCertificate>E9999999</p:SalesTaxCertificate>
      <p:TaxID>123-89-4567</p:TaxID>
      <p:StateOfIncorporation>VIC</p:StateOfIncorporation>
      <p:Status>Active</p:Status>
    </p:Customer>
    <p:OrderHeader>
      <p:OrderType>ST</p:OrderType>
      <p:OrderNumber>00099999</p:OrderNumber>
      <p:Currency>AU</Currency>
      <p:DeliveryLocation>
        OCS CENTRAL WAREHOUSE ADDRESS
      </p:DeliveryLocation>
      <p:OrderTerms>NET FROM STATEMENT 30 DAYS</p:OrderTerms>
      <p:OrderTotals>
        <p:FreightTotal>0.00</p:FreightTotal>
        <p:OnCostTotal>0.00</p:OnCostTotal>
        <p:OrderTotal>39.50</p:OrderTotal>
      </p:OrderTotals>
    </p:OrderHeader>
    <p:OrderLines>
      <p:Line>
        <p:LineNumber>0001</p:LineNumber>
        <p:Quantity>1.0000</p:Quantity>
        <p:LineValue>21.00</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
      </p:Line>
      <p:Line>
        <p:LineNumber>0002</p:LineNumber>
        <p:Quantity>50.0000</p:Quantity>
        <p:LineValue>18.50</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
      </p:Line>
      <p:Line>
        <p:LineNumber>0003</p:LineNumber>
        <p:Quantity>30.0000</p:Quantity>
        <p:LineValue>457.50</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
      </p:Line>
      <p:Line>
        <p:LineNumber>0004</p:LineNumber>
        <p:Quantity>20.0000</p:Quantity>
        <p:LineValue>025.08</p:LineValue>
        <p:Status>Active</p:Status>
        <p:PackingInstructions/>
      </p:Line>
    </p:OrderLines>
  </p:PurchaseOrder>
</Response>
```

the service provider, we now move on to discuss how the request is routed to a network-based cache for processing.

5.2.2. Making a Routing Decision. Once all required fragments and cache metadata are represented in the response, a routing decision must be made—it is time to decide how the message should be sent to ensure that all fragments are obtained before the final message is sent on to the consumer.

First, the provider application (specifically, the custom XML serialization library) identifies a coordinating router (CR). This router is responsible for not only serving fragments from its local cache but also for contacting other neighborhood cache-enabled routers that contain needed fragments that are not stored in the CR's cache. The set of neighborhood cache-enabled routers is determined by limiting the set of those routers within v (typically $v = 1$ or 2) hops of the CR. This limitation is imposed to prevent retrieving cache fragments from distant caches at high costs.

The choice of the coordinating router in our scheme is based on optimizing a cost function C_{CR} that attempts to quantify the cost of routing the service message response from the service provider to the service consumer through the CR. This cost incorporates the following components: (a) the cost of transporting the provider response to the CR, (b) the cost of acquiring fragments from the local cache and neighborhood nodes, and (c) the cost of transporting the final response to the service consumer. The cost function C_{CR} is computed as follows:

$$\begin{aligned} C_{CR} = {} & f(\text{size of response from the provider, network cost from provider to the CR}); \\ & + f(\text{size of useful fragments available at routers in the neighborhood of the CR, network cost from the CR to these neighborhood routers}); \\ & + f(\text{size of final response, network cost from the CR to the requestor}), \\ & \text{where } f(a,b) = a \times b. \end{aligned} \quad (1)$$

This function will always generate a CR except in the case where none of the needed fragments are stored on the network (or all fragments are stored too distantly). In this case, because there are no fragments to `get`, all caching instructions will be `set` calls, which do not require a CR.

In our scheme, we will always pick the node that has the minimum cost. If two nodes have the same cost, we break the tie by picking the one that is nearest to the requestor in terms of the number of network hops.

Once the CR is identified, the library removes all nonneighborhood routers listed in any `get` fragment's `ValidList`. If any `get` fragment is left with an empty `ValidList`, the fragment element is removed from the `get` list, the corresponding object is retrieved from cache, and the fragment content is generated through standard XML serialization processing and inserted into the response message. At this point, the message is ready to be sent to the CR.

Next, we describe the processing that occurs in the coordinating router.

5.2.3. Processing on a Cache-Enabled Router. When a cache-enabled XML router receives a response message containing a `CachingInstructions` element, it stores all cacheable fragments contained in the message (following the `set` instructions) to its local cache, regardless of whether it acts as the CR for this message. Through this mechanism, every cache-enabled router that "sees" a cache-enabled message will store any available elements to its local cache. The provider application, with the help of the caching library, monitors the network "ack" messages that are returned for every network hop made by the message on its way to the intended recipient. For each such "ack" received from a cache-enabled router, the provider updates its cache directory to note that the cacheable fragment set in the message is stored in that router's cache.

`Get` instructions are processed only by the CR. When a cache-enabled message arrives at the designated CR, the router first retrieves any elements stored in its local cache, inserts them into the response, and determines which other needed fragments are not stored locally. The CR then contacts its neighboring XML routers (up to v hops away) based on each `Get` instruction's `ValidList` to request these missing fragments and inserts them into the response message upon receiving them. These fragments are also stored to the CR's local cache. At this point, the message is complete. After the CR removes the caching instructions from the message, the CR sends it off to the service consumer.

5.3. Cache Maintenance

In our approach, we assume the presence of a general purpose distributed main memory cache, which provides basic cache management functionality including basic invalidation and any required distributed synchronization. A detailed discussion of cache management in distributed cache scenarios is covered extensively in the literature (Fan et al. 2000, Tewari et al. 1999).

In this section, we discuss the specifics of cache maintenance for our fragment caching scheme. These differ slightly from the requirements of a general purpose distributed caching scheme. Specifically, we describe a cache replacement policy for a router-based cache to handle the case where a new fragment arrives

and must be stored but the space allocated for cache storage is already full. Here, when a router must evict an old fragment to make room for a new one, the router follows a configurable replacement policy, e.g., least-recently used (LRU), first in, first out (FIFO), etc. (many such policies have been described in the literature, for example, Podlipnig and Bszrmenyi 2003), and *selects a fragment for eviction from the same provider as that of the incoming fragment*. Selecting a fragment from the same provider helps ensure that no single provider's fragments dominate cache storage on a given cache-enabled router.

When a replacement event occurs on a cache, the router notifies the provider and waits for the provider to acknowledge the eviction. The provider updates its cache directory (removes the fragment from the `ValidList` in cache) and sends back an acknowledgement to the router. Only then does the router actually delete the element from cache. This prevents race conditions, e.g., where the provider sends a `Get` instruction to a router where the corresponding fragment has been evicted from cache through replacement processing.

The expiration of a cached element's ttl triggers invalidation of the fragment in all cache storage locations. Once an item becomes invalid on the provider, it will not be returned from cache and therefore cannot be assigned a `Get` instruction at caching instruction insertion time. Invalid fragments are prioritized over valid fragments for cache replacement processing purposes.

5.4. Availability and Fault Tolerance

In this section, we discuss how we handle unexpected situations where a `Get` instruction for an XON router cannot be fulfilled because of router unavailability.

There are two ways a router can go down: (a) with notification to the service provider, or (b) without notification to the service provider. Case (a) occurs when the router catches an exception (e.g., try catch processing) and notifies all service provider caches that it is shutting down. In this case, no service provider will generate `Get` caching instructions for fragments on that router.

In case (b), where the router did not have the opportunity to notify the service providers about its unavailability, the service provider will not receive the periodic "heartbeat" messages from the router and will mark it as "unavailable." The service provider will not assign any further responsibility to an "unavailable" router until it rejoins the network and begins broadcasting "heartbeat" messages again.

These two cases cover the vast majority of potential race conditions related to router availability. However, it is still possible that a service provider could dispatch a `Get` instruction for fragments on (or assign CR responsibilities to) an unavailable router while a shutdown notification traverses the network (for case (a)) or within the heartbeat period (case (b)).

If a CR receives a `Get` instruction for a fragment that is only valid on an unavailable router, it will notify the service provider of the router's unavailability and request the content from the service provider (available in the service provider cache) via a separate service.

If a router goes down after it has been assigned CR responsibilities by a service provider, the message will bounce back to the service provider indicating the router's unavailability. In this case, the service provider will retrieve the content from its local cache and send it directly to the consumer.

For cases where a router must request content from a service provider, there will be a small delay in providing the response to the client; however, the probability of such race conditions overall is low so this happens only very infrequently. On average, the total system will perform much better than in the no-cache case so the benefits accrued from caching on the network far outweigh the costs associated with handling such faults.

6. Analytical Study

As in other caching systems, end-to-end response time and bandwidth consumption are the two areas where benefits accrue. In this section, we first analytically study how our caching scheme impacts bandwidth consumption and end-to-end response time. We then present a study of the space and time complexity of our scheme. Table 7 shows a list of notations used in our analysis.

6.1. Benefits of Our Caching Scheme

In this section, we model bandwidth usage and response time for our caching scheme mathematically. In §7, we experimentally demonstrate the benefits of caching under several scenarios.

6.1.1. Bandwidth Consumption. In the no-cache case, a service message passes through the XML routers in the overlay network without change so the bandwidth usage for every XML network hop remains constant. In contrast, in our caching scheme, a cache-enabled XML message passes through the CR, an intermediate node in the XML overlay network that inserts the cached fragments to complete the message to be delivered to the client. Thus, in the cache case, the bandwidth consumption before the CR differs from the bandwidth consumption after the CR.

To accurately compare the no-cache and cache cases, we compute the average bandwidth consumption per hop in the network. First, we compute the network bandwidth consumption at each hop from the server

Table 7 Table of Notation

Symbol	Description
$\mathcal{R} = \{R_1, R_2, \ldots, R_m\}$	Responses
$\Gamma = \{F_1, \ldots, F_k\}$	Fragments
SF_i	Average size of a fragment F_i
fR_j	Number of fragments in a response R_j
$E_j = \{e_i \mid e_i \in \Gamma, i = 1, \ldots, fR_j\}$	Fragment set for a response type R_j
$\mathrm{SR}_j = \sum_{i \in E_j} SF_i$	Average size of response type R_j
f	Average regular header size
f_c	Size of the additional header content per cacheable fragment required for caching
$l_{a,b}$	Number of XML routers between router a and b
lookup	Cache table lookup time on the provider
δ	Average transmission time per byte between two hops in the network
r	XML routing computation and caching instruction generation time
$\mathrm{CF}_i = k_1\,\mathrm{SF}_i$	Average access time for fragment F_i to read from or write to a cache
$sw_i = k_2\,\mathrm{SF}_i$	Average switching time of fragments of type F_i in XML stack
pF_i	Average server generation time of fragments of type F_i
B_{hop}	Bandwidth consumption per hop
b_u	Number of bytes flowing over the uth hop in a network path
Δ_u	Number of bytes inserted by the source node of the uth hop in a network path
m_k	Number of occurrences of the kth fragment type in a response
RT_i	Response time for the response R_i
rt_j	Portion of the response time contributed by the jth distinct fragment in R_i
P_j	Probability that fragment j is cached in a particular routing node
h	Hit ratio in the cache
Z	Fraction of total fragments cached
ACK	Number of bytes for acknowledgement message for each fragment in the cache
INV	Number of bytes for invalidation message for each fragment in the cache

to the client in the XML overlay network, and then we average it across all these hops to compute the average per-hop bandwidth consumption in the network in the cache case.

Let B_{hop} denote the bandwidth consumption per hop. We assume that each hop is of equal length (or weight). Thus, we represent B_{hop} as follows:

$$B_{\mathrm{hop}} = \frac{\sum_{u=0\ldots U-1} b_u}{U}, \tag{2}$$

where u represents the uth hop on the delivery path of the response from the origin server O to destination node T. There are a total of $U = l_{O,T}$ hops, $0 \ldots (U-1)$, on the path and b_u is the number of bytes flowing over the uth hop. The bandwidth consumption at each hop, b_u, for this particular request can be computed as follows in Equation (3), where Δ_u is the payload (fragments) inserted by the source node of hop u of the path:

$$b_u = b_{u-1} + \Delta_u, \quad \forall\, u = 1, \ldots, U-1. \tag{3}$$

For the first hop of the path, $b_0 = f + f_c + \Delta_0$ for $u = 0$, where f is the fixed header size, f_c is the size of the extra header content that enables caching, and Δ_0 is the payload inserted by the origin server. The value of Δ_0 will depend on the fragment set that needs to be generated at the server or inserted from the provider's local cache.

At any router in the XML overlay network, $\Delta_u = 0$ if the response only passes through the source node of hop u with no processing; otherwise, $\Delta_u = \sum_k \mathrm{SF}_k\, m_k$, where SF_k is the size of the kth distinct fragment added by the source node of hop u and m_k is the number of occurrences of that fragment in the response. Thus, B_{hop} is then generally rewritten as given in Equation (4):

$$B_{\mathrm{hop}} = (f + f_c + \Delta_0) + \Delta_1 \frac{(U-1)}{U} + \Delta_2 \frac{(U-2)}{U} + \cdots + \Delta_{U-1} \frac{(U-1)}{U}. \tag{4}$$

Clearly, the CR's fetching of fragments introduces extra traffic on the network. If the CR needs to fetch fragments, the associated additional byte transmission overhead is $\sum_y \sum_k \mathrm{SF}_{yk}$, where y indexes the yth node that CR has to fetch fragments from, k indexes the kth distinct fragment that CR fetches from node N_y, and SF_{yk} is the size of the kth fragment that CR fetches from node N_y.

Where there is at most one CR node between the provider O and the destination T, B_{hop} is

$$B_{\mathrm{hop}}^{\mathrm{cache}} = \frac{f_c l_{O,T} + (f + \Delta_O)U + \Delta_{\mathrm{CR}} l_{CR,T} + \sum_y \sum_k \mathrm{SF}_{yk}\, l_{\mathrm{CR},N_y}}{U + \sum_y l_{\mathrm{CR},N_y}} + \sum_k \mathrm{ACK} + \sum_k \mathrm{INV}, \tag{5}$$

where the first term denotes the overhead because of cache tags, the second term denotes the bandwidth size coming out of the provider O without cache tags, the third term denotes the bandwidth that is being added by the CR, the fourth term denotes the overhead because of the CR accumulating fragments from other nodes N_y, the fifth term denotes the bandwidth overhead because of the acknowledgement message, and the sixth term denotes the bandwidth overhead because of the invalidation message. Also, note that $l_{O,\mathrm{CR}} + l_{\mathrm{CR},T} = l_{O,T} = U$. The Δ_O and Δ_{CR} are computed as follows:

$$\Delta_O = \sum_{k \mid F_k \text{ is generated by the service provider}} \mathrm{SF}_k \tag{6}$$

$$\Delta_{\mathrm{CR}} = \sum_{k \mid F_k \text{ is provided by the CR}} \mathrm{SF}_k . \tag{7}$$

In the no-cache case, the bandwidth consumption at each hop from the provider O to the destination T remains unchanged. Thus, for the no-cache case, we have

$$B_{\mathrm{hop}}^{\mathrm{no\_cache}} = f + \sum_{k \mid F_k \text{ is generated by provider}} \mathrm{SF}_k . \tag{8}$$

If we consider h as the hit ratio and assume that Z is the fraction of the fragments cached, we compute the total B_{hop} for all fragments for the cache case as follows:

$$B_{\mathrm{hop}}^{\mathrm{total}} = ZhB_{\mathrm{hop}}^{\mathrm{cache}} + (1 - Zh)B_{\mathrm{hop}}^{\mathrm{no\_cache}}. \tag{9}$$

Clearly, the benefit of caching will happen when

$$B_{\mathrm{hop}}^{\mathrm{cache}} < B_{\mathrm{hop}}^{\mathrm{no\_cache}}, \tag{10}$$

i.e.,

$$f_c l_{O,T} + \sum_y \sum_k \mathrm{SF}_{yk}\, l_{\mathrm{CR},N_y} + \left(U + \sum_y l_{\mathrm{CR},N_y}\right) \cdot \sum_k (\mathrm{ACK} + \mathrm{INV}) < \Delta_{\mathrm{CR}} l_{O,\mathrm{CR}}. \tag{11}$$

We noted earlier that if the CR needs to fetch fragments from neighboring routers, this introduces a cost that reduces the benefit of caching on the network. We explore this trade-off by considering a set of typical parameter values based on an analysis of the messaging characteristics and the XML network setup at a major retail organization (these parameter values will also be used in our simulation study in §7.3). This company has deployed a service-oriented architecture to achieve seamless communication across multiple departments to handle various activities such as order handling, purchasing processes, inventory management, and customer support. It thus serves as a strong basis for our experimental model. Table 8 shows the parameter values.

Table 8 Parameter Values

Parameter	Values ranges	Distribution
Number of requests	200, 500, 1,000	Not applicable
Number of fragments	10,000, 25,000, 50,000	Not applicable
Number of routers	Baseline value 30	Constant
SF_i	Baseline value 15,000–25,000 bytes	Uniform
fR_j	40–60	Uniform
E_j	Fragments are randomly assigned to requests	
$\mathrm{SR}_j = \sum_{i \in E_j} \mathrm{SF}_i$	Computed	Not applicable
f	1,024	Constant
f_c	512–1,024 Bytes	Uniform
$l_{a,b}$	10–20	Uniform
lookup	10 ms	Not applicable
r	2 ms	Constant
k_1	2 ms	Constant
k_2	2 ms	Constant
pF_i	Baseline value 400–600 ms	Uniform
h	Baseline value 0.8	Constant
Percent of cacheable fragments	Baseline value 80%	Constant
δ	10 Mbps	Constant
ACK	100 bytes	Constant
INV	100 bytes	Constant
Network connectivity	Base line value 20%	Constant

We take the average values of each of these parameters and a hit ratio of 0.4. By plugging these values into Equation (11), we find that caching will be beneficial if

$$4{,}768 l_{O,T} + 404{,}000 \bar{l}_{\mathrm{CR},N_y} < 400{,}000 l_{O,\mathrm{CR}}, \tag{12}$$

where $\bar{l}_{\mathrm{CR},N_y}$ denotes the average distance (in number of hops) between the CR and other routers from which fragments must be fetched. If all of the necessary fragments are fetched from the CR, then $l_{\mathrm{CR},N_y} = 0$ and clearly the caching is beneficial.

If the routers from which the CR fetches fragments are far away from the CR (i.e., the distance comparable to the distance between the CR and the origin node $l_{O,\mathrm{CR}}$), the benefit of caching in the network is lost by the overhead of fetching the content from other nodes to the CR. More specifically, if $\bar{l}_{\mathrm{CR},N_y}$ and $l_{O,\mathrm{CR}}$ have values that are very close to one another, then caching is not beneficial. In our implementation of the algorithm, we consider only those nodes that are within a one-hop distance from the CR (i.e., $\bar{l}_{\mathrm{CR},N_y} = 1$). Otherwise, the content is drawn from the cache at the service provider.

6.1.2. Response Time. In this section, we analytically compare the response time for the cache case with that of the no-cache case. First, we derive the response time for the cache case.

Let RT_i be the response time for response R_i. $\mathrm{RT}_i = \sum_j rt_j$, where rt_j represents the portion of the response time contributed by the jth distinct fragment of R_i. rt_j can be decomposed into three parts:

1. Time for the service provider (origin O) to produce fragment F_j: rt_j^O.
2. Time for the CR to fetch the fragment F_j: rt_j^{CR}.
3. Transmission time of F_j from origin O to destination T: $\mathrm{Tr}_{O,T}$.

Next, we derive each of the parts that comprise the overall response time. In deriving these times, we consider the time required to store a fragment into the cache as negligible because this typically happens asynchronously. We also assume that the time required to transmit a byte across one hop in the network is same for all XML network hops.

(a) Time for the producer to produce fragment F_j. Let X_j be a variable, $X_j = 1$ if F_j is cacheable, and $X_j = 0$ if F_j is not cacheable:

$$rt_j^O = (1 - X_j)(pF_j + sw_j) + X_j\big(hr + (1-h)(pF_j + sw_j)\big). \tag{13}$$

When F_j is not cacheable, the sender needs to compute the fragment pF_j and pass it through the XML stack sw_j. When F_j is cacheable and available from a cache on the network, the origin node generates the appropriate header and performs the routing computation r. If F_j is cacheable but not available in the cache, the sender computes the fragment pF_j and passes it through the XML stack.

(b) Time for the CR to produce fragment F_j. Let Y_j be a variable, and $Y_j = 1$ if F_j is cached in the CR or 0 if F_j is cached in a node other than the CR:

$$rt_j^{\mathrm{CR}} = X_j h\big(sw_j + lookup + \mathrm{CF}_j + (1 - Y_j)(\delta\,\mathrm{SF}_j\, l_{\mathrm{CR},N_j})\big). \tag{14}$$

No matter where F_j is cached, the component retrieving F_j has to perform a cache lookup (*lookup*), retrieve the fragment CF_j, and pass it through XML stack sw_j. If it is cached in a place other than the CR, then transmission time is needed to fetch the fragment from the node where it is currently cached (N_j) to the CR. Here, δ is transmission time per byte per hop.

(c) Transmission time of F_j from O to T.

$$\begin{aligned}\mathrm{Tr}_{O,T} = {} & (1 - X_j) l_{O,T}\,\mathrm{SF}_j\,\delta \\ & + X_j\big(h l_{\mathrm{CR},T}\,\mathrm{SF}_j\,\delta + (1-h) l_{O,T}\,\mathrm{SF}_j\,\delta + f_c l_{O,T}\delta\big) \\ & + l_{O,T} f\delta.\end{aligned} \tag{15}$$

If the fragment is not cacheable, the entire fragment F_j of size SF_j will be transmitted from the origin O to the receiver T (expressed in the first term). This is also the case for cacheable fragments not stored in any cache (expressed in the third term). If there is a cache hit, the fragment will be transmitted from the CR to the receiver (expressed in the second term).

Adding these three parts, we have the following:

$$\begin{aligned}rt_j^{\mathrm{cache}} = {} & (1 - X_j)\big(pF_j + sw_j + l_{O,T}\,\mathrm{SF}_j\,\delta\big) \\ & + X_j h\big(r + sw_j + lookup + \mathrm{CF}_j + (1 - Y_j)\delta\,\mathrm{SF}_j\, l_{\mathrm{CR},N_j} \\ & \qquad + l_{\mathrm{CR},T}\,\mathrm{SF}_j\,\delta\big) \\ & + X_j(1-h)(pF_j + sw_j + l_{O,T}\,\mathrm{SF}_j\,\delta) \\ & + X_j f_c l_{O,T}\delta + l_{O,T} f\delta.\end{aligned} \tag{16}$$

In the no-cache case, the complete fragment F_j of the response RT is generated at the origin O and is transmitted through the XML network to the receiver T:

$$rt_j^{\mathrm{no\_cache}} = pF_j + sw_j + l_{O,T}\,\mathrm{SF}_j\,\delta + l_{O,T} f\delta. \tag{17}$$

Thus, the benefit of caching in the response time accrues when the following inequality is satisfied, where P_j is the probability that j is cached in CR (note that $l_{O,T} = l_{O,\mathrm{CR}} + l_{\mathrm{CR},T}$):

$$\begin{aligned}&\sum_j \big(P_j(r + lookup + \mathrm{CF}_j + f_c l_{O,T}\delta) + (1 - P_j)\delta\,\mathrm{SF}_j\, l_{\mathrm{CR},N_j}\big) \\ &\quad < \sum_j \big(pF_j + l_{O,\mathrm{CR}}\,\mathrm{SF}_j\,\delta\big).\end{aligned} \tag{18}$$

Using average values from Table 8 and the 0.4 hit ratio, from Equation (18) we get

$$\begin{aligned}&P_j(22 + 0.768 l_{O,T}) + (1 - P_j)20\bar{l}_{\mathrm{CR},N_y} \\ &\qquad < 500 + 20 l_{O,\mathrm{CR}},\end{aligned} \tag{19}$$

where $\bar{l}_{\mathrm{CR},N_y}$ is the average number of hops between the coordinating node and the nodes from where the cached fragments are fetched.

Following the discussion in §6.1.1, we assume $\bar{l}_{\mathrm{CR},N_y} = 1$, i.e., the CR will fetch cacheable fragments only from nodes that are one hop away from the CR. We also assume that 50% of the cached contents are fetched from the coordinating node, i.e., $P_j = 0.5$. The network fragment caching will be beneficial if following inequality holds:

$$0.384 l_{O,T} < 499 + 20 l_{O,\mathrm{CR}}. \tag{20}$$

The above inequality will hold true in all scenarios, except where $l_{O,T}$ is a very large value and $l_{O,\mathrm{CR}}$ is very small, i.e., when the distance between the origin node and the destination node is very large and the CR is very close to the origin node. In such scenarios, the effect of caching in the CR is not beneficial and only the service provider cache should be used.

6.2. Caching Scheme Complexity

The space complexity needed by the service provider is, in the worst case, an LM matrix for the cache directory, where L is the number of the XML routers existing in the XML router fabric and M is the number of cacheable fragments of all its service message responses. Each matrix element contains the size, ttl, ID, available in cache, and content of the fragment if the fragment is cached and valid in the particular XML router (we note, though, that the content can be stored once rather than for each fragment-router pair). The space complexity of a CR is measured as the size of the cached fragments of the service message responses that pass through it plus the size of the cache directory of the IDs of all the fragments it stores.

In terms of time complexity, a provider will need to:

(a) process the request to determine which fragments need to be generated and drawn from cache $O(fR)$;

(b) select the best CR with a cache lookup $O(1)$, a computation to find the cost for each router $O(L)$, and selection of the best CR in logarithmic time $O(\log(L))$; and

(c) generate the response, which takes time $O(1)$.

Thus, the total time complexity at the provider for the cache algorithm is $O(L+\log(L))$.

The CR will need to contact a neighboring set of routers for needed fragments. This requires, in the worst case, the number of <get> fragments in a response $O(fR)$.

7. Experimental Results

We performed a simulation study to validate the expected benefits of our method and present a set of those results in this section. In this study, we considered the impacts of our caching scheme in terms of both bandwidth and response time as well as the scalability of the scheme in terms of the CPU usage required for the cache algorithm.

We consider two scenarios: (a) a cache case that represents the use of our proposed service message fragment caching scheme; and (b) a no-cache case in which caching is not enabled. To demonstrate the efficacy of the scheme, we are interested in two metrics: (a) bandwidth savings (expressed as the ratio of cache bandwidth usage to no-cache bandwidth usage) and (b) improvement in end-to-end response time (expressed as the ratio of cache response time to no-cache response time).

The simulation was written in Java and run on Windows XP Pentium 2.1 GHz machines with 4 GB RAM. In the simulation, we assigned baseline values to various parameters based on an analysis of the messaging characteristics and the network setup at a major retail organization. These parameter value ranges are shown in Table 8. In this company, about 500 different types of service messages are exchanged among applications. Through a combination of automatic and manual analysis, we identified about 2,500 fragment types. Of these, about 80% are cacheable and the remainder change too frequently to merit caching.

The company's deployment consists of 30 XML routers that handle SOAP-level security and message routing among other tasks. These 30 XML routers are connected by a hybrid ring-and-tree topology. An analysis of this topology revealed about 20% network connectivity.

We assume a Zipfian distribution in which 80% of the fragments are cached by 20% of the routers as is common in Web and Internet experiments (Breslau et al. 1999).

We simulate a network of XML routers created by randomly connecting each router in the network with a predefined percentage of routers in the network. We call this predefined percentage *network connectivity*. The baseline value of such network connectivity is 20%, i.e., each router is directly connected to 20% of the routers in the network.

The simulation involves a multithreaded Java program. Each thread of this program represents a node in the network. Each thread keeps track of the contents of its local cache in an in-memory data structure. The program precreates a network of nodes to simulate the XML network as a graph structure. There is a one-to-one mapping between these nodes and the Java threads. The program preselects two nodes as the destination and the origin. The origin thread keeps track of which cache fragments reside in which nodes (i.e., the cache table) and how the network is connected by a routing table.

The simulation program generates a service message composed of several fragments following power law distribution, i.e., in 80% of cases, 20% of fragments are accessed. This distribution is based on the typical behavior found in Web application usage (Breslau et al. 1999).

The routing of the message from the origin to the destination is selected by running the routing algorithm and by looking at the cache table, which keeps track of which nodes store which cached fragments.

Once the origin node thread has created the initial message, it passes the message to the CR as selected by the routing algorithm and the path defined based on the routing table. The CR adds the cached content and passes the full service message to the destination node following the path defined in the routing table.

At each step, the network delays, the fragment generation delays, and the lookup cost delay are introduced by inducing sleep time in the thread

corresponding to each node. The size of each data exchange across any two threads is measured in bytes. At the end of the experiment, the bandwidth usage across all hops is summed to compute the bandwidth for the cache case. Similarly, the response time is computed by measuring the time difference between when the origin thread starts creating the response message and when the destination thread receives the response, including the running time of the routing algorithm. In the no-cache case, the routing algorithm does not run; the response content is created at the origin node and passed to the destination node based on the routing table. The network delay and fragment generation costs are introduced by sleep time in the appropriate threads. Again, the bandwidth and the response time are measured by summing usage across hops and total delays, respectively.

Each simulation experiment was run for a total of 10,000 requests. We compute the average bandwidth utilization at the provider site and the end-to-end response time at the service consumer site across these 10,000 requests.

We present a total of eight experimental results in considering the impacts of caching. Four consider bandwidth impact as *fragment size*, *hit ratio*, *percentage of cacheable fragments*, and *network connectivity* are varied. Four consider response time impact as *percentage of cacheable fragments*, *hit ratio*, *average processing time at the provider*, and *network connectivity* are varied.

Each of these experiments was run for each value of request count and fragment count. For each combination of request count and fragment count, the simulation experiment was run 3 times, with a different random seed for each run, for a total of 27 simulation experiments in each case.

Because the values of each of the independent parameters (average fragment size, percent cacheable fragments, hit ratio, fragment size, network connectivity, and average processing time) are varied, the values of other parameters are kept at baseline values as indicated in Table 8.

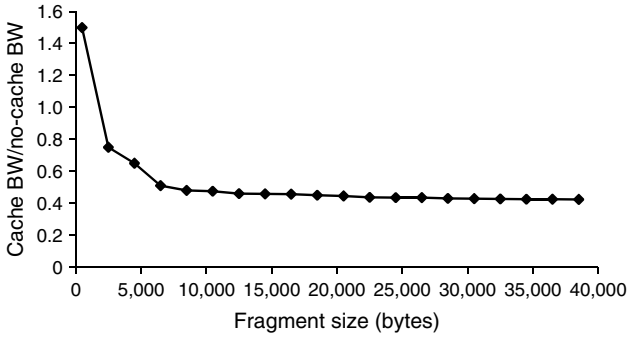
We also experimentally demonstrate the scalability of our caching scheme in terms of CPU usage as the *average number of fragments* and the *number of routers* are varied. These results were gathered during experimental runs for the other experiments in our simulation study.

7.1. Bandwidth

In this section, we demonstrate how bandwidth consumption varies as the independent parameters *fragment size*, *hit ratio*, *percentage of cacheable fragments*, and *network connectivity* are varied.

7.1.1. Fragment Size. Figure 12 demonstrates how the ratio of bandwidth usage between the cache case and the no-cache case (cache BW/no-cache BW) varies as the average fragment size is increased from 500 bytes to 40,000 bytes. The point of this experiment is to determine the effect of the byte overhead of caching metadata in the cache case to the full-size content overhead in the no-cache case.

Figure 12 **Variation of Bandwidth with Fragment Size**

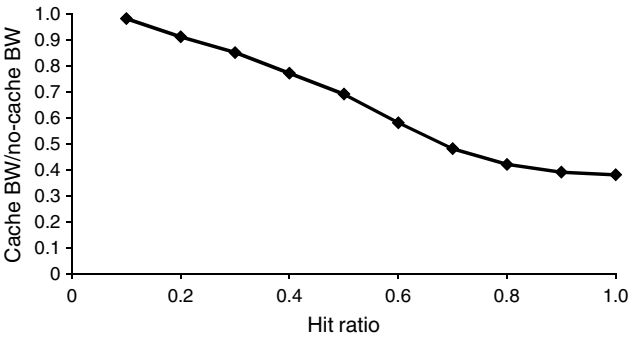


As the average fragment size is increased from 500 bytes to 8,500 bytes, the ratio decreases dramatically from 1.5 to 0.48. At higher byte sizes, the ratio decreases more slowly with increases in the average fragment size. The ratio reaches 0.42 when the fragment size is 40,000 bytes.

Initially, the overhead from adding cache tags is comparable to fragment size so the benefit of caching is not significant. For smaller fragment sizes, as the fragment size is increased, the reduction of overhead from cache tags as a percentage of fragment size is high and results in the initial sharp decrease in the cache BW/no-cache BW ratio. For larger fragment sizes, the reduction of overhead in terms of the percentage of the total fragment size is very small. Recall that the baseline hit ratio is 80% and the baseline percentage of cacheable fragments is 80% so the provider still needs to generate the uncached and uncacheable content. Overhead related to generated content and metadata accounts for the remaining portion of the cache BW/no-cache BW ratio.

7.1.2. Hit Ratio. Figure 13 demonstrates how bandwidth savings increase as the hit ratio increases. At a low hit ratio of 0.1, bandwidth usage in the cache

Figure 13 **Variation of Bandwidth with Hit Ratio**



case is almost equal to the no-cache bandwidth—98%. As the hit ratio increases, the ratio cache BW/no-cache BW decreases. At a hit ratio of 0.8, the cache BW is approximately 42% of the no-cache BW. This is because of the effect of increasing hit ratio—as more fragments can be drawn from cache, the bandwidth requirements are reduced.

7.1.3. Percentage of Cacheable Fragments. Figure 14 demonstrates how the ratio cache BW/no-cache BW varies along with the percentage of cacheable fragments. As the percentage of cacheable fragments among all fragments is increased, the bandwidth savings in the cache case is increased and the ratio of cache BW/no-cache BW decreases. At 10% cacheable fragments, the bandwidth in the cache case is 88% of the no-cache case—a 12% improvement in bandwidth. In the ideal case of 100% cacheable fragments, the bandwidth in the cache case is 32% of the no-cache case. This remaining 32% comprises uncached content (the baseline hit ratio is 80% in these experiments so not all objects will be found in cache even in the case of 100% cacheable objects) and caching metadata.

7.1.4. Network Connectivity. Figure 15 demonstrates how the ratio cache BW/no-cache BW varies along with the network connectivity. A higher value of network connectivity indicates that the network of routers is densely connected while a lower value indicates a more sparsely connected network.

Overall, the range of difference shown in Figure 15 is approximately 0.1. At 20% network connectivity, i.e., when each router is directly connected to 20% of the other routers in the network, the ratio is 0.42. At 50% connectivity, the ratio is 0.48, and at 100% connectivity, i.e., when each router is directly connected with all other routers, the ratio is 0.52. At 100% network connectivity, the benefit of caching is mostly derived from avoiding the cost of generating the fragments.

As network connectivity increases (i.e., as a network becomes more densely connected), the cache BW/no-cache BW ratio increases, which indicates that the benefits of caching are greater in more sparsely connected networks (most networks are sparsely connected) than in more densely connected networks. This occurs because messages require fewer hops in more densely connected networks (and therefore less bandwidth) to reach their destinations in both the cache and no-cache cases. Because bandwidth costs are lower overall in more densely connected networks, there is less room for improvement through caching.

Figure 14 **Variation of Bandwidth with Percentage Cacheable Fragments**

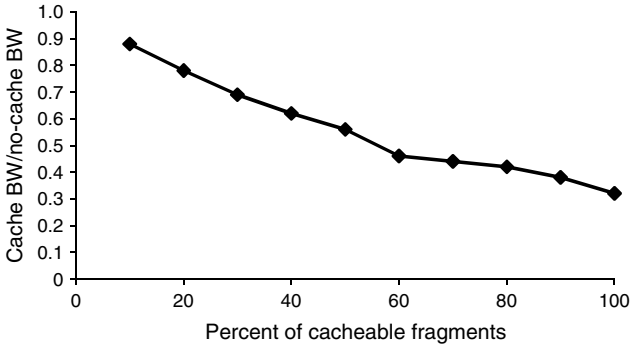
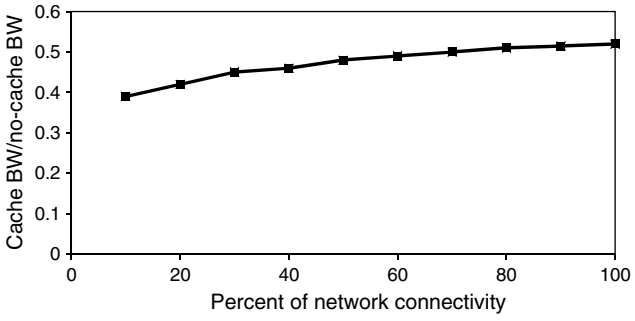


Figure 15 **Variation of Bandwidth with Network Connectivity**

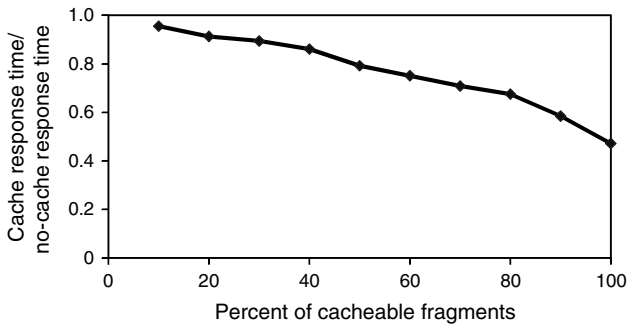


7.2. Response Time

In this section, we demonstrate how the end-to-end response time of the application varies as the independent parameters *percentage of cacheable fragments*, *hit ratio*, *average processing time at the provider*, and *network connectivity* are varied.

7.2.1. Percentage of Cacheable Fragments. Figure 16 shows how the cache/no-cache response time ratio varies along with the percentage of cacheable fragments. As the percentage of cacheable fragments is increased, more fragments can be fetched from the cache, saving processing time and transmission time associated with these fragments. This results in increased improvements in response time for the cache case. When only 10% of fragments are cacheable, the ratio is 0.96. In the best case, when

Figure 16 **Variation of Response Time with Percentage Cacheable Fragments**



100% of the fragments are cacheable, the ratio is approximately 0.47. (Recall that the hit ratio remains at 80% in these experiments so some content must still be generated for every request to handle uncached content.)

7.2.2. Hit Ratio. Figure 17 shows how the cache/no-cache response time ratio varies with increases in the hit ratio. The increased hit ratio results in decreased end-to-end response times because of savings in processing costs on the provider. However, one interesting point to note is that at a very low hit ratio of 0.1, the response time in the cache case is actually worse than in the no-cache case. This is because of the fact that at low hit ratios, there is overhead in checking to see whether the fragment instance is available in the cache. The approximate threshold hit ratio where the cache becomes beneficial in terms of response time is $h = 0.15$. We note that a 15% hit ratio is highly achievable. In practice, hit ratios as high as 40%–50% (or higher) are commonly achieved in many Web caching scenarios if good choices are made for caching granularity, time to live, and primary-secondary cache size (Caceres et al. 1998, Cao and Irani 1997, Squid Project 2010, Nikolov 2009).

7.2.3. Average Processing Time. Figure 18 shows how the cache/no-cache response time ratio varies with increases in the average processing time at the provider side to generate fragments. As the average processing time increases, the benefit of caching increases, resulting in decreased end-to-end response times. However, at a very low processing times (on the order of 10 milliseconds), the overhead of caching (the cache lookup time, switching cost in the router, etc.) is larger than the processing time. Thus, there is no benefit of caching at very low processing times; rather, the response time is actually worse when caching is enabled in these cases.

7.2.4. Network Connectivity. Figure 19 demonstrates how the cache/no-cache response time ratio varies as network connectivity is varied. The results shown here are similar to those of the bandwidth network connectivity experiments. At higher network connectivity percentages (more densely connected networks), a response must traverse fewer nodes on the way to its destination, which means that the overall response time is lower in both the cache and no-cache cases. Thus, in all cases, there is less room for improvement through caching as the network becomes more densely connected, which leads to a higher ratio at higher densities of connectivity.

7.3. Scalability

In this section, we demonstrate how our routing algorithm scales with the number of fragments per response message and the number of routers. Figure 20 demonstrates the percentage of CPU required

Figure 17 Variation of Response Time with Percentage Hit Ratio

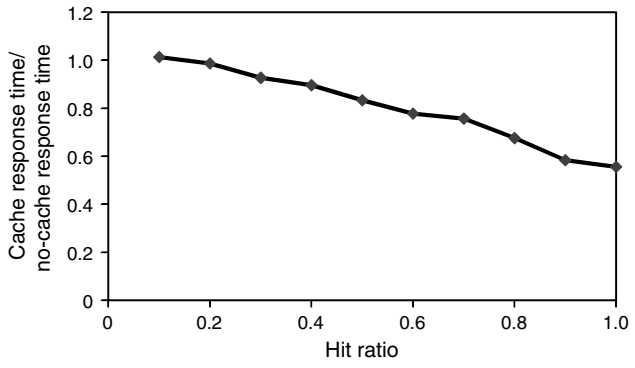


Figure 18 Variation of Response Time with Average Processing Time

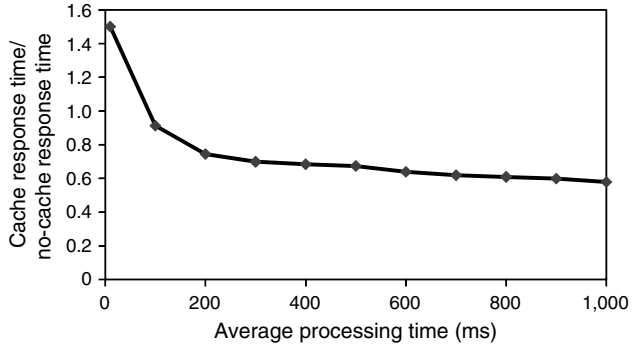


Figure 19 Response Time Improvement with Network Connectivity

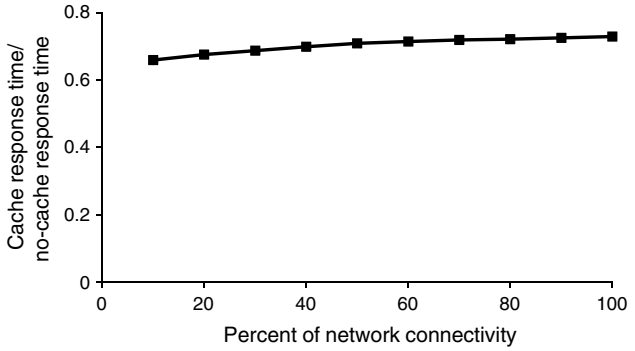


Figure 20 CPU Overhead with Number of Fragments Per Response

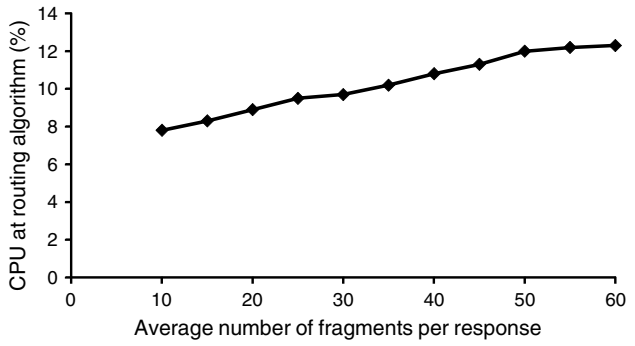
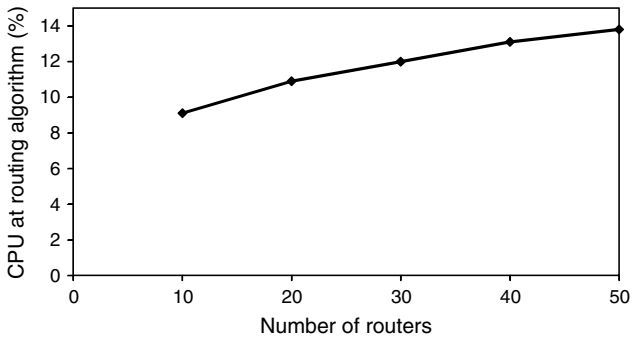


Figure 21 CPU Overhead with Number of Routers in the Overlay Network



to run the routing algorithm on a 2.1 GHz Pentium processor machine. The CPU overhead increases almost linearly with the number of fragments. When a response has about 60 fragments, the CPU overhead is 12%. Figure 21 demonstrates the CPU required to run the routing algorithm along with a number of routers. Clearly, the CPU overhead increases linearly with the number of routers. In an enterprise XML overlay network, the typical number of routers will be in the range of a few tens of routers. At 50 routers, the CPU overhead is just 13.5%.

8. Case Study

In this section, we report on the integration of our approach into a commercial XON router and a set of experiments describing the performance of our approach in a staging environment.

8.1. Commercial Integration

We describe in detail our implementation of XML fragment caching as integrated into a commercial application server middleware software (enabling caching in the service provider) and one of the most popular XML routers, the Cisco Application-Oriented Network Module (AON), which runs on the Cisco 2600 router (Cisco Systems, Inc. 2010).

8.1.1. Integrating Caching and Routing Logic at the Service Provider. We integrated our caching logic into the Tomcat application server[2] software for the purposes of this case study, although we could follow this integration approach for any middleware software. As shown in Figure 10, we add two components to a service provider to enable caching: a cache and a custom XML serialization library. We describe each of these components in turn.

The cache is an external process, main memory cache that supports basic caching functionality. Caching logic is accessed from within the service provider through a cache client library, implemented in Java, that provides access to the cache through API calls, e.g., `check-cache` and `get-cache` (this implementation is similar to that discussed in Datta et al. 2004).

The custom serialization library performs standard XML serialization (creating XML-formatted versions of application objects) and provides the additional functionality required to add caching instructions and perform routing decision making. Further, we instrumented the custom library to maintain counts of fragments served from cache or generated by the back end server in order to track the hit ratio for experimental purposes. We wrote our own custom XML serialization library (in Java, of course), although it would be possible to modify any standard XML serialization API (such as the Java JavaScript Object Notation (JSON) library) for our purposes.

8.1.2. Integrating Caching Logic at the XON Router. We first describe the AON architecture in general and then discuss how we integrated our caching logic into the AON module.

The AON module is a Java component running in a Tomcat environment on the BSD UNIX Operating System on the Cisco 2600 router. This module provides multiple application-oriented services at the XML network layer such as XSLT transformation, authentication, compression, caching services, content lookup, encryption, and message logging. These services are implemented as components called bladelets. When an XML message arrives at the AON router, it passes through multiple bladelets depending on the services the message requires. The AON module is extensible; it allows the inclusion of custom bladelets with user-defined functionality. Such bladelets are developed in Java using the AON API.

We implemented our fragment caching system as a custom bladelet. Following the AON architecture, the bladelet is written in Java. The bladelet uses AON's built-in cache service API to store and retrieve XML fragment content in the AON cache. The cache service of the AON router also manages the synchronization and invalidation of fragment contents among multiple AON instances. The details of this cache service are part of past research work (Datta et al. 2004) and are described in detail in Cisco Systems, Inc. (2008).

8.2. Experiments

We performed a set of experiments on our fragment cache implementation in a staging environment. We describe the experimental setup and results in this section.

Our testing environment consisted of a network of five AON routers configured with our caching logic as described in §§8.1.1 and 8.1.2. These AON routers connect two Tomcat application server instances running on Linux/Java servers. One of these server instances acts as the origin node (i.e., the service

[2] http://tomcat.apache.org/.

provider) while the other is the destination node (i.e., the service consumer). Both the service provider and service consumer servers are 2.1 GHz dual-core Intel Processor Windows XP machines with 8 GB RAM. On the service provider, 1 GB of RAM is reserved for in-memory caching. On the AON routers, 512 MB is reserved for in-memory caching.

To simulate a realistic application at the origin node, we configured part of the TPC-App benchmark (Transaction Processing Performance Council 2010), which is composed of a database schema of tables `Item`, `Stock`, `Author`, `Customer`, `Orders`, `Order_line`, `Address`, and `Country`. Based on this schema, the destination node (service consumer) selects an object from this list along with its primary key value (selected following the Zipfian distribution), and then sends a request for the object along with all its related objects in XML message format from the origin node. For example, if the destination node requests an entity from the `orders` table, the service provider should return all nested objects in the order, e.g., `Order_line`, `Customer`, `Address`, `Country`, `Item`, and `Author`. The details of the schema and the table description can be found in the TPC-App description (Transaction Processing Performance Council 2010).

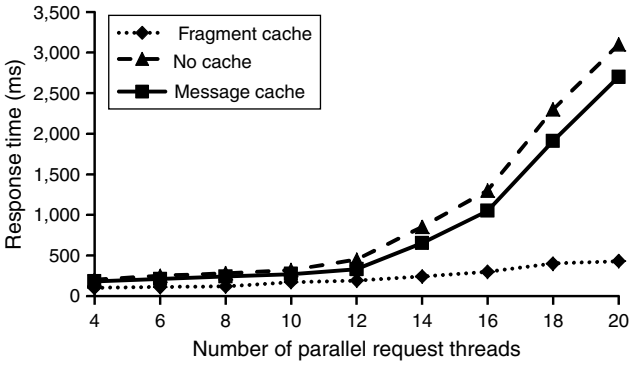
The total database size of the TPC-App benchmark we used was 1 GB. Each cached object is stored in serialized object form on the service provider and in its XML format (similar to the example scenario) on the AON routers. The destination node generates requests at a periodic interval from multiple parallel threads. The number of parallel threads from which requests emanate is gradually increased in our experiments to generate additional request loads.

In our first experimental case, we measure the average time for the round-trip request-response message from the destination node to origin and back (response time) as well as the average number of requests per second completed (throughput) with our fragment caching scheme in place. We plot both the response time and throughput results against the number of parallel request threads in Figures 22 and 23, respectively.

We repeated the same experiment along with the AON router network but without the fragment cache bladelet and the caching infrastructure on the service provider. These results are also plotted along with the fragment cache results in Figures 22 and 23.

We repeated the same experiment again for full message caching. The AON router contains a message cache bladelet that, when enabled, stores request-response pairs. When configured across the entire AON router network, all the routers through which a message traverses will store request-response pairs. If a request arrives at an AON router and the router finds that it exactly matches a request stored in cache, the router will deliver the XML response from the message cache without passing the request to the back end server. Response time and throughput results for these experiments are also shown in Figures 22 and 23, respectively. Further, we added a small custom bladelet designed to count messages delivered from cache on each router for hit ratio tracking purposes in our experiments, and we report hit ratio comparison results for the fragment cache and message cache cases.

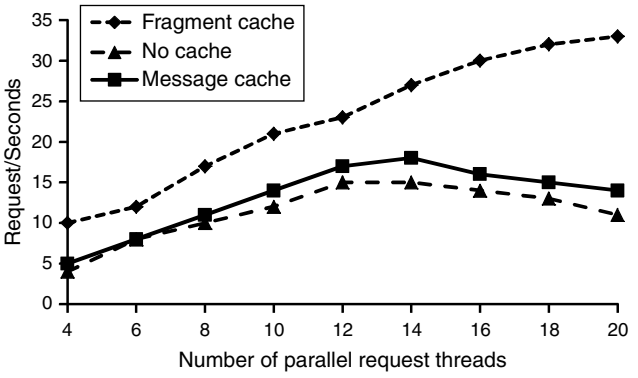
Figure 22 Variation of Response Time with Number of Threads



To highlight the differences between the fragment cache, message cache, and no-cache scenarios in these experiments, we treated every object as cacheable, i.e., the percentage of cacheable fragments was 100% (the expected effects of varying the percentage of cacheable fragments are described in §7.3).

We first consider hit ratio results for the fragment cache and message cache cases. We found that the average hit ratio for the fragment cache scenario was around 0.5 (i.e., approximately half of the fragments were delivered from network-based caches) while the average hit ratio across all routers in the message cache case was only 0.04 (i.e., approximately 4% of messages were served from cache). The difference reflects the level of reusability of stored data in each case—for a full message to be reusable from cache,

Figure 23 Variation of Throughput with Number of Threads



the exact same request must pass through a router where its corresponding response is stored in cache. That is, only 4% of messages in this experiment could be drawn from the full message cache because only 4% of cached messages contained the exact content required for service calls. When the number of possible requests is very large, the probability of such matching is low.

Though there may be overlapping fragments within the messages stored in the full message cache, there is no way to identify and reuse overlapping fragments when messages are stored in full message form and it is not possible to take advantage of any potential for reuse.

In contrast, our fragment caching method is able to take advantage of that potential for reuse and serve cacheable fragments into service messages on the network. Because content is cached at a finer granularity than in the full message caching case, fragments are reusable across a larger percentage of responses, which results in a higher hit ratio (50%) for the fragment cache case.

The implication of this difference in hit ratio is that 96% of service requests in the full message caching case could not be serviced from the network-based cache. For each cache miss, the request proceeded to the provider, where the provider did all of the work required to generate the full message (including service logic and XML serialization). In the fragment caching case, the provider needed to generate and serialize only 50% of needed fragments, with the remainder served from network-based caches. More work is required to generate and serialize fragments than is required to insert needed fragments into service messages at run time. Use of this information resulted in significant improvements in response time performance for the fragment caching case in our experiments.

We next consider our response time results. Figure 22 reports the response time results for the fragment cache, message cache, and no-cache cases as the number of parallel request threads increases. In general, the response time for the message cache case is slightly lower than that of the no-cache case, but the fragment cache case response time is much lower than the message cache case throughout the experiments.

At lower numbers of parallel request threads, the fragment cache case performance is about two times better (i.e., roughly half the average response time) compared to the message cache and no-cache cases, and the performance results in the message cache and no-cache cases are virtually equivalent.

As the number of parallel request threads increases, the fragment cache case scales much better than both the no-cache and message cache cases. This results in response time performance in the fragment cache case that is far better than those of both no-cache and message cache cases at higher numbers of parallel threads. This is demonstrated by the significant growth rate of the response time curve in no-cache cases starting at 12 parallel threads. The response time curve for message cache cases shows similar behavior beyond 14 parallel threads. At higher numbers of parallel threads, the application server and database processing must generate each and every object and serialize these objects into XML format. This creates substantial performance bottlenecks for both no-cache cases (where each request must be fully processed) and message cache cases (where responses must be fully generated for each request that reaches the service provider—approximately 96% of requests in these experiments). In the fragment cache case, the complete response object does not need to be serialized; rather, only the primary key needs to be written along with its `get` tag in the XML message, which mitigates a significant portion of the work performed in the no-cache and message cache cases. Although there is a marginal improvement in response time in the message cache case compared to the no-cache case, the improvement is not that significant when compared with the fragment cache case.

Figure 23 reports the throughput results for these experiments. Here, the significant feature to note is in the no-cache and the message cache curves—noncache throughput peaks at 12 parallel request threads (the same thread count where response time began to increase dramatically in Figure 22) and the message cache throughput peaks at 14 parallel threads (the same thread count where the response time began to increase dramatically for the message cache case in Figure 22). These are the points where the no-cache and message cache cases reach their respective peak capacities; adding load beyond this point actually results in reduced throughput. Here, we see that the message cache case shows a marginal improvement in performance over the no-cache case, as demonstrated by the difference in their respective bottleneck points (12 parallel threads in the no-cache case as compared to 14 parallel threads in the message cache case). In contrast to both the no-cache and message cache cases, the fragment cache case continues to handle increased loads to higher request thread counts. In fact, the fragment cache case did not reach its peak capacity in our experiments.

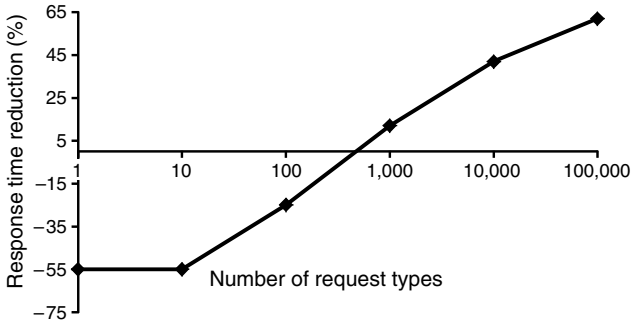
We ran a further experiment to more fully contrast the fragment cache and message cache cases. We ran this experiment using the same experimental testbed described for the previous experiment and using the same code modules for the fragment cache and message cache implementations. In this experiment, we varied the number of possible unique responses by varying the number of unique requests that can be

generated by a service consumer, and then we measured the end-to-end response time at the service consumer in both the fragment cache and message cache cases across multiple set size cases. In Figure 24, we report the percentage savings in response time in the fragment cache case compared to the message cache case as the number of possible unique responses increases. We note that the horizonal axis in Figure 24 is in logarithmic scale.

When the number of possible unique responses is low, the probability that the corresponding XML response will be found in full message cache is high. When every response is being served directly from a full message cache, the request need not go to the service provider at all. In contrast, in the fragment cache case, even if all needed fragments are served from routers, the request still needs to pass through the service provider to identify the needed fragments. When all the unique responses and their corresponding responses can be cached in the message cache case, we would expect the full message cache to perform better than the fragment cache. As the number of possible unique responses rises, fewer of the needed full message responses are likely to be found in message caches. In contrast, fragments have a higher rate of reusability than full messages; we would expect that in such scenarios the fragment cache would outperform full-message caching.

We show our results for this experiment in Figure 24. At 1 to 10 unique responses, the hit ratio in the message cache case was nearly 100%; here, the message cache case outperformed the fragment cache case. At 100 unique responses, the message cache case still outperformed the fragment cache, though the hit ratio was no longer 100%. This resulted in a narrowing of the performance difference between the message cache and fragment cache cases. However, when the number of unique responses increased to 1,000 or more, we started to see the benefit of the fragment cache, which marginally outperformed the message cache at this unique response count. At high numbers of unique responses (say, 10,000), the probability of a cache hit in the message cache case is very low; in this case, virtually all responses are served from the provider with corresponding decreases in performance in comparison to the fragment caching case. At very high numbers of unique responses (say, 100,000), the full message cache has virtually no utility; here, we see a significant performance improvement for the fragment cache case as compared to the message cache case.

Figure 24 **Percentage of Savings in Response Time in Fragment Cache Compared to Message Cache**



From these results, we can conclude that in scenarios where the number of possible unique responses is low, a message cache approach is more beneficial than a fragment cache approach. However, in a complex enterprise system where the number of possible unique responses is very large, the message cache case provides no benefit. In such scenarios, as we have demonstrated in Figures 22 and 23, a fragment cache will provide greater benefits in terms of improved response time and throughput performance.

9. Discussion and Managerial Implications

From a practical perspective, this work provides several relevant insights for IT managers who have implemented SOAs. We discuss these insights here.

First, we have proposed a workable scheme for utilizing existing XON network fabrics to support caching for service-oriented message content. This enables managers to derive desirable benefits, e.g., improved SOA performance in terms of end user response time and bandwidth usage as well as improved quality of service in terms of better service level agreement (SLA) performance. If an XON is already in place and nominal CPU is available on the routers, these benefits can be garnered without additional infrastructure investment.

Second, rolling out the caching functionality requires little implementation effort on the part of the IT unit. The serverside and routerside caching algorithms are implemented as plug-ins, which add little CPU overhead at run time and thus scale well with network size. IT managers need only specify (using the caching API) which elements are cacheable and which attribute(s) uniquely identify each cacheable element. This enables the caching of service message content at a granularity that is more likely to be reused than full message caching.

Third, the results of our analysis and experiments reveal several insights relevant to caching decision making for an IT manager considering implementing network-based caching for service message content. We summarize these insights here.

- The average size of a cached service message fragment must be large enough to make the overhead of the addition of cache headers worthwhile. For

very small service message fragments, caching will actually result in higher costs than without caching because of the overheads introduced by the cache header content. As the average size of a service message fragment increases, the addition of cache header content overhead becomes a better investment because of the increases in bandwidth savings at larger fragment sizes.

• The expected frequency of reuse is an important consideration for determining a service message element's cacheability. For a service message element to be worth caching, there should be a reasonable expectation that it will be accessed again within the element's ttl period. The higher the frequency of access for a service message element within its ttl, the better it is for caching because the response time and bandwidth benefits of caching only accrue with reuse.

• In a similar vein, the percentage of service message fragment content that is cacheable as compared to noncacheable service message content must be considered. When only a very small percentage of service message content is cacheable, there are fewer opportunities to reap the benefits of service message caching. As the percentage of cacheable content rises, the benefits that will accrue from caching rise as well. If there is significant reusability of full messages, then full message caching using traditional HTML message caching techniques can be applied.

• The density of network connectivity is also an important factor to consider for this caching scheme. In a sparsely connected network, a message requires more hops to reach its destination than in a more densely connected network. In such scenarios, caching will provide greater bandwidth savings and better response time performance as compared to more densely connected networks. This is because additional caching benefits accrue with every network hop that is made by a service message, and messages traverse more hops on average in more sparsely connected networks compared to more densely connected networks. Though caching will still provide improved bandwidth utilization and response time performance in densely connected networks, the benefits are more pronounced in more sparsely connected networks.

There are a number of considerations that managers need to contemplate in the course of deploying a solution of the type we propose here. These considerations fall into two categories: (a) what should be cached; and (b) what to consider in terms of trust, privacy, and security.

The question of what to cache is an interesting one and there are a number of dimensions to keep in mind. First, we must consider the nature of the data that is to be cached. Certain sensitive data may raise legal or privacy issues. Enabling security controls can alleviate much of the risk of caching such data. However, there may be cases where standard security controls do not sufficiently reduce the privacy or legal risks. Here, caching should be avoided.

Second, a candidate fragment for caching should meet several important criteria. We should be able to expect to reuse a fragment, perhaps multiple times, before the underlying data is expected to change (e.g., exact inventory for a particular item may be updated frequently because orders are continuously being fulfilled). Some data elements change more frequently than they would be reused; these should not be cached because the work of storing them would be wasted.

For candidate fragments meeting these criteria, the nested nature of the response XML in a service message can make it difficult to decide what exactly should be cached. To illustrate this complexity, we return to our purchase order example. Suppose that customer purchase data may be aggregated by a different service on the same service provider such that all items (cacheable fragments used in other services) purchased by the customer over the course of a billing period (e.g., a month) are aggregated for use by accounts payable in a `<Billable>` element. The question then arises: Should we cache billable purchase data by individual purchase order or should we aggregate it by billing cycle? Answering this question requires that one consider trade-offs across a number of criteria including the expected reusability of fragments at each granularity. This is complex and beyond the scope of our work here. We note, however, that a similar question is addressed (Dutta et al. 2006) in the context of caching for in-memory application objects; managers can find useful insights into such trade-offs in that work.

Finally, we consider the question of how we can ensure that the recipient should be able to receive cached content—i.e., issues of security, privacy, and trust. As a foundation, the SOAP protocol and underlying application services have provisions for authentication and access control. When enabled, these ensure (with or without caching) that the service message recipient and the identity it claimed are the same (authentication), and that the recipient is allowed to access the functions and data in its request (access control). Coupled with the use of secure transport protocols, basic SOA infrastructures can be secured against inappropriate access. Trust (e.g., that a service provider is the provider it claims to be) can be enabled with similar authentication mechanisms.

We consider more closely the question of security in the context of our proposed scheme. The issue of security for the caching scheme comes down to ensuring that the cache locations on the network are secure,

and this is really a matter of ensuring that appropriate authentication and access control schemes are in place on the physical router to prevent inappropriate access. XML routers are equipped, much like server operating systems, with fully functional authentication and access control functionalities that are similar to those provided by major operating systems.

Having considered the insights this work can provide to managers and discussed considerations such as what to cache and how to ensure security and trust, we move on to concluding our paper.

10. Conclusion

In this paper, we proposed a caching scheme for service response message fragments in service-oriented architectures. We provided background overviews of XML-formatted service request/response message formats and the XML router technology underlying our scheme, and we presented a detailed overview of the architecture and processing of our approach. We developed an analysis of the bandwidth and response time impacts of our method and ran a detailed simulation study to validate the benefits of our scheme. Our simulation experiment results showed up to 60% bandwidth reductions and 50% improvement in response time with low CPU overhead. Further, in our case study, we integrated our caching approach into a commercial XON router and an off-the-shelf application server middleware software. Our experiments using the case study integration demonstrate that when there is no resource bottleneck, the fragment cache-enabled case reduces average response times by 40%–50% and increases throughput by 150% compared to the no-cache and full message caching cases. In experiments contrasting fragment caching and full message caching, we found that full message caching provides benefits when the number of possible unique requests is low, and we also find that the benefits of fragment caching increase as the number of unique requests increases. These experimental results clearly demonstrate the benefits of our approach.

Acknowledgments

The authors gratefully acknowledge comments from Senior Editor Debabrata Dey and Associate Editor Giri Kumar Tayi as well as the reviewers. Their feedback has helped significantly improve this manuscript. The authors are listed alphabetically. Each author contributed equally to this effort. This work was done while Kaushik Dutta was at Florida International University.

References

Anthias, T., K. Sankar. 2006. The network's new role. *ACM Queue* **4**(4) 38–46.

Apache Jakarta Java Caching System Project. 2010. Java caching system. Accessed February 12, 2010, http://jakarta.apache.org/jcs/UsingJCSBasicWeb.html.

Azim, O., A. K. Hamid. 2002. Cache SOAP services on the client side. Accessed October 1, 2010, http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap.html.

Breslau, L., P. Cao, L. Fan, G. Phillips, S. Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. *Proc. Conf. Comput. Comm., New York*, 126–134.

Caceres, R., F. Douglis, A. Feldmann, G. Glass, M. Rabinovich. 1998. Web proxy caching: The devil is in the details. *ACM SIGMETRICS Performance Evaluation Rev.* **26**(3) 11–15.

Cao, P., S. Irani. 1997. Greedydual-size: A cost-aware WWW proxy caching algorithm. *2nd Web Caching Workshop, Boulder, CO*.

Chen, L., E. A. Rundensteiner, S. Wang. 2002. XCache-A semantic caching system for XML queries. *Proc. ACM SIGMOD—Demo, Madison, WI*, 618.

Chidlovskii, B., U. M. Borghoff. 2000. Semantic caching of Web queries. *VLDB J.* **9**(1) 2–17.

Cisco Systems, Inc. 2008. Cisco AON Programming Guide, 3.0. Cisco Systems, Inc., http://www.cisco.com/applilcation/pdf/en/us/guest/products/ps6480/c1667/ccmigration_09186a008-08054b6.pdf.

Cisco Systems, Inc. 2010. Cisco AON product. Accessed February 12, 2010, http://www.cisco.com/en/US/products/ps6449/index.html.

Cohen, F. 2006. *Fast SOA: The Way to Use Native XML Technology to Achieve Service Oriented Architecture Governance, Scalability, and Performance*. Morgan Kaufmann, San Francisco.

Dar, S., M. J. Franklin, B. T. Jónsson, D. Srivastava, M. Tan. 1996. Semantic data caching and replacement. *Proc. VLDB, Bombay, India*, 330–341.

Datta, A., K. Dutta, H. M. Thomas, D. E VanderMeer. 2003. World wide wait: A study of Internet scalability and cache-based approaches to alleviate it. *Management Sci.* **49**(10) 1425–1444.

Datta, A., K. Dutta, H. Thomas, D. VanderMeer, K. Ramamritham. 2004. Proxy-based acceleration of dynamically generated content on the World Wide Web: An approach and implementation. *ACM Trans. Database Systems* (*TODS*) **29**(2) 403–443.

Dattani, K., M. Pandit, M. Zirn. 2010. Data tier caching for SOA performance. Accessed October 1, 2010, http://www.oracle.com/technology/architect/enterprise_solution_cookbook/data_tier_caching_for_soa.html.

Dortch, M. 2008. Performance in a service oriented architecture world. Analyst report, Aberdeen Group, Boston. http://www.aberdeen.com/Aberdeen-Library/4198/RA-performance-service-architecture.aspx.

Dutta, K., S. Soni, S. Narasimhan, A. Datta. 2006. Optimization in object caching. *INFORMS J. Comput.* **18**(2) 243–254.

Fan, L., P. Cao, J. Almeida, A. Z. Broder. 2000. Summary cache: A scalable wide-area Web cache sharing protocol. *IEEE/ACM Trans. Networking* **8**(3) 281–293.

Fenner, W., M. Rabinovich, K. K. Ramakrishnan, D. Srivastava, Y. Zhang. 2005. XTreeNet: Scalable overlay networks for XML content distribution and querying. *Proc. 10th Internat. Workshop Web Content Caching Distribution, Sophia Antipolis, French Riviera, France*, 69–80.

Gruman, G. 2005. Countrywide Financial simplifies lending. InfoWorld. Accessed August 4, 2009, http://www.infoworld.com/article/05/05/02/18FEsoa-countrywide_1.html.

Hericko, M., M. B. Juric, I. Rozman, S. Beloglavec, A. Zivkovic. 2003. Object serialization analysis and comparison in Java and .net. *ACM SIGPLAN Notices* **38**(8) 44–54.

Hosanagar, K., R. Krishnan, J. Chuang, V. Choudhary. 2005. Pricing and resource allocation in caching services with multiple levels of quality of service. *Management Sci.* **52**(12) 1844–1859.

IBM. 2006. RouteOne secures credit application management system. Accessed August 4, 2009, ftp://ftp.software.ibm.com/software/websphere/integration/datapower/R1_CaseStudy.pdf.

Koch, C. 2007. Reaping the big business benefits of SOA. Accessed August 4, 2009, http://www.cio.com/article/print/121952.

Lelli, F., G. Maron, S. Orlando. 2006. Improving the performance of XML based technologies by caching and reusing information. *Internat. Conf. Web Services*, Salt Lake City, UT, 689–700.

Liang, Q. A., J.-Y. Chung, H. Lei. 2006. Service discovery in P2P service-oriented environments. *8th IEEE Internat. Conf. E-Commerce Tech. and 3rd IEEE Internat. Conf. E-Commerce, E-Services, Enterprise Comput., San Francisco*, 46–53.

Liang, Q., X. Wu, H. C. Lau. 2009. Optimizing service systems based on application-level QOS. *IEEE Trans. Services Comput.* **2**(2) 108–121.

Limaye, B. 2005. Implementing a cluster-aware cache using WebLogic. WebLogic Developer's Journal. Accessed February 12, 2010, http://weblogic.sys-con.com/node/102681.

Mimoso, M. S. 2004. XML straining network performance, bandwidth. Accessed February 12, 2010, http://searchsoa.techtarget.com/news/article/0,289142,sid26_gci1028139,00.html.

Mookerjee, V. S., Y. Tan. 2002. Analysis of a least recently used cache management policy for Web browsers. *Oper. Res.* **50**(2) 345–357.

Nikolov, A. V. 2009. Effects of the coherency on the performance of the Web cache proxy server. *Internat. J. Comput. Sci. Network Security* **9**(4) 158–162.

Olston, C., A. Manjhi, C. Garrod, A. Ailamaki, B. M. Maggs, T. C. Mowry. 2005. A scalability service for dynamic Web applications. *Proc. Conf. Innovative Data Systems Res., Asilomar, CA*, 56–69.

OpenSymphony OSCache Project. 2010. OS cache usage. Accessed February 12, 2010, http://www.opensymphony.com/oscache/wiki/API%20Usage.html.

Oracle Corporation. 2010. Oracle WebLogic v10.3 documentation. Accessed February 12, 2010, http://download.oracle.com/docs/cd/E13155_01/wlp/docs-103/javadoc/com/bea/p13n/cache/Cache.html.

Podlipnig, S., L. Böszörmenyi. 2003. A survey of Web cache replacement strategies. *ACM Comput. Surveys* **35**(4) 374–398.

Powell, M. 2002. XML Web service caching strategies. Accessed October 1, 2010, http://msdn.microsoft.com/en-us/library/aa480499.aspx.

Research and Markets, Inc. 2003. XML processing and wirespeed messaging market opportunities, strategies, and forecasts. Research and Markets research report, Dublin, Ireland. http://www.wintergreenresearch.com/reports/XML_Final.htm.

Rogers, S. 2007. Worldwide SOA-driven software 2007–2011 forecast: A changing IT lifestyle. Technical Report 207080, IDC, Framingham, MA. http://www.sap.com/corporate-en/press.epx?pressid=8433, http://www.marketresearch.com/product/display.asp?productid=1517202.

Seltzsam, S., R. Holzhauser, A. Kemper. 2005. Semantic caching for Web services. *Proc. Service-Oriented Computing—ICSOC, Amsterdam*, 324–340.

Squid Project. 2010. Squid: Optimising Web delivery. Accessed February 12, 2010, http://www.squid-cache.org/.

Tatemura, J., A. Sawires, O. Po, D. Agrawal, K. S. Candan. 2005. Wrex: A scalable middleware architecture to enable XML caching for Web services. *Proc. ACM/IFIP/USENIX 6th Internat. Middleware Conf., Grenoble, France*, 124–143.

Tewari, R., M. Dahlin, H. M. Vin, J. S. Kay. 1999. Design considerations for distributed caching on the Internet. *Internat. Conf. Distributed Comput. Systems, Columbus, OH*, 273–284.

Tilkov, S. 2009. *REST und HTTP: Einsatz der Architektur des Web fr Integrationsszenarien*, Chapter 10. dpunkt Verlag. http://www.dpunkt.de/buecher/3493/rest-und-http.html.

Transaction Processing Performance Council. 2010. TPC-App benchmark. Accessed February 12, 2010, http://www.tpc.org/tpc_app/default.asp.

Verton, D. 2003. U.S. Navy formalizes XML management. ComputerWorld. Accessed August 4, 2009, http://www.computerworld.com/developmenttopics/development/xml/story/0,10801,79348,00.html.

Wu, J., Q. Liang, E. Bertino. 2009. Improving scalability of software cloud for composite Web services. *IEEE Internat. Conf. Cloud Comput., Bangalore, India*, 143–146.

Yang, L. H., M.-L. Lee, W. Hsu. 2003. Efficient mining of XML query patterns for caching. *Proc. 29th Internat. Conf. VLDB, Berlin*, 69–80.