RJ 10327 (A0311-052) November 13, 2003
Computer Science

IBM Research Report

Database Support for Web Service Caching

Jussi Myllymaki, Berthold Reinwald
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

Database Support for Web Service Caching

Jussi Myllymaki
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA
jussi@almaden.ibm.com

Berthold Reinwald
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, USA
reinwald@almaden.ibm.com

ABSTRACT

Web Service protocols are a key component in enabling businesses to link applications together within and across enterprise networks. However, the extensibility of the protocols and the use of XML as a high-level data format create formidable challenges in terms of application performance. Also, system availability is bound to deteriorate as the use of Web Services over wide area networks expands. As a result of the increasing interdependence of applications, the concept of a "weakest link" becomes that much more apparent.

One approach to mitigating these performance and availability problems is to cache responses from Web Service requests. In this paper we discuss requirements for Web Service caching and outline a plan for adding caching support to a database system. The paper focuses on the semantics of caching–what data can be cached, what can be reused, and how Web Service protocols can be extended to better support caching.

1. INTRODUCTION

Web Service protocols are a key component in enabling businesses to link applications together within and across enterprise networks. Not only do Web Services make application integration and collaboration easier by leveraging a platform-neutral protocol and agreed-upon XML format, they ease application development by offering greater freedom in the choice of tools and platforms. Web Services also help mitigate system evolution problems such as those caused by changing APIs and version upgrades.

The concept of Web Services may mean different things to different people—ranging from a software development and tooling paradigm, to an open platform for multi-vendor application collaboration, to user-oriented services (e.g. Web calendaring) offered by Internet service providers. A common view of Web Services, also adopted in our work, is that of an extensible, open *protocol* that enables applications to talk to each other and exchange business data, e.g. in a supply chain across enterprise networks. In essence, Web Services enable integration of business processes.

The extensibility of Web Service protocols and the use of XML as a high-level data format create formidable challenges in terms of application performance. The performance problem is partly due to the high latency of wide-area network connections [22] (e.g. HTTP) and partly due to the high cost of XML parsing and schema validation.

At the same time, system availability deteriorates as the number of participating systems and layers (network routers, gateways, application servers, etc.) grows. In the end, the overall availability of the system can only be as good as that of the "weakest link."

One approach to mitigating performance and availability problems is to cache as much of the wide-area network communication as possible. In the case of Web Services and SOAP [16] in particular, this translates to caching responses from SOAP requests. Given that SOAP requests typically use HTTP as the transport protocol and that HTTP caching is already done on the Internet, a natural approach is to try to extend the existing HTTP caching mechanism (proxy caches and HTTP headers) with SOAP support. However, since SOAP requests are POSTed over HTTP, not all request data is visible to an HTTP proxy, for instance SOAP method call parameters and message headers containing expiration information [18]. As a result, existing HTTP proxy caching mechanisms are not adequate for Web Service caching.

Relational database products, including IBM DB2 [9, 7], Oracle [11], and Microsoft SQL Server 2000 [12], have adopted Web Services in two ways: by exposing database functionality (e.g. stored procedures and user-defined functions) as Web Services (*provider*) and by integrating external Web Services into federated databases (*consumer*).

Our work extends these efforts by integrating caching into a Web Service-aware database system at a very low level, thereby providing business applications built on top of the database with a high-performance Web Service interface. In this paper we discuss requirements for effective and practical Web Service caching and outline a plan for adding caching support to a database system. Our discussion focuses on the semantics of caching—what data can be cached, what can be reused, and how Web Service protocols can be extended to better support caching.

The paper is organized as follows. In Section 2, we review existing work on Web caching and coherence models. We discuss requirements for Web Service caching in Section 3 and present an outline of database-supported Web Service caching in Section 4. Results of a performance study on a prototype implementation of a Web Service cache are presented in Section 5. Extending the cache with an invalidation protocol is described in Section 6. A summary and directions for future work appear in Section 7.

2. RELATED WORK

An extensive body of work exists on cache coherence models and cooperative caches, generally arising from the database and distributed systems fields. Research on Web caching and consistency has also seen interest, whereas Web Services caching is only now starting to gain attraction.

2.1 Cache Coherence

Cache coherence models are typically based on one of three mechanisms: expiration times, polling, and invalidations. In each category, several possibilities exist. For instance, many different methods exist for estimating a document's time-to-live (TTL) or time-to-refresh (TTR). TTL may be fixed if the data has that inherent characteristic and the value is known. Some dynamic and semi-dynamic TTL schemes are based on the speed at which changes occur and the most recent TTL values [21], as old values may be workable predictors for future values. An adaptive TTR value may also be subject to static upper/lower bounds and the highest frequency of changes observed so far.

In polling, a client contacts the server periodically, at document expiration time, or on every request (poll-every-time) to determine if a refresh is needed. Some protocols, e.g. HTTP with its If-Modified-Since request modifier, have explicit support for making polling cheaper than a full retrieval. More frequent polling translates to less stale data and stronger coherence. Combinations of client pull and server push have also been studied [3]. For instance, if a server knows the TTR value of a client, it can preemptively push data updates to the client before the next refresh.

Invalidation aims to eliminate staleness by requiring that a server inform all clients that cache its data whenever some of the data has changed and must be refreshed or invalidated. Leases are a hybrid of TTL and invalidation schemes [5]. A client acquires a lease for a specific amount of time, during which the cache can read the data and the server must request permission before attempting to change it. However, once the lease expires, the server regains control of the data with no further client-server communication required.

A major drawback of invalidations and leases is the extra bookkeeping a server must do and the extra cost of client-server communication. An opportunistic lease renewal scheme [1] reduces the communication cost of lease renewal and, as a result, can reduce the length of leases. The authors suggest reductions in lease length by a factor of 50 with only a 1% network overhead. The reduction is primarily due to using one lease per client computer, not per data object cached, as "computers or networks are expected to fail, not individual data objects." [1] To ensure coherence, however, leases must be supplemented by explicit data locks.

A survey of cache consistency mechanisms in use on the Internet [6] found that weak consistency (allowing staleness) reduced network traffic more than either TTL or invalidation and can be tuned to return stale data less than 5% of the time. They also found that files tended to have bimodal lifetimes—either they were modified very frequently or very infrequently.

A qualitative comparison of caching strategies for Web application servers is shown in [14]. The author notes that the advantage of push mechanisms is the reduced response time for first hit and lower cost of updating data. In contrast, pull methods use cache space more effectively, as data isn't retrieved into the cache until it is accessed, which also helps ensure that only the hottest data stays in the cache.

2.2 Cooperative Caches

Harvest [2] is hierarchical Internet cache typically configured as a tree. A server with a cache miss can request data from its siblings and parent using unicast messages. The authors note that OS and DBMS caching is very different from Internet caching in that in the former a cache miss may cost several orders of magnitude more than a cache hit, but in the latter only one order of magnitude more.

The need to minimize not only cache hit cost but also cache miss cost is highlighted in [22] where three design principles for large-scale distributed caches are discussed: minimize the number of hops to access data on both hits and misses, share data among many users and scale to many caches, and cache data close to clients. We note that our approach for caching Web Service responses is aligned with these requirements—data is cached inside the database where it is close to database applications and shared by many of them.

Realizing that coordination and communication effort increases as a function of the number of caches and may actually decrease the overall effectiveness of caching, an optimal degree of cooperation and topology for disseminating updates has been studied [20]. A hierarchical topology closely resembling that of the DNS structure has also been proposed [8]. Each DNS domain maps to a cache neighborhood containing at most one copy of a file. A multicast-based adaptive caching scheme [23] delivers the same data to multiple receivers and also acts as an information discovery vehicle. A directory of the locations of cached data is replaced by a multicast query to nearby caches.

In a large-scale content distribution network (CDN), none of the existing coherence mechanisms (TTL, client polling, invalidation, adaptive refresh, and leases) may be sufficient [15]. The authors propose a cooperative lease mechanism and Δ-consistency semantics and allow multiple proxies to share a single lease. Multicast is used to propagate server notifications to caches, which reduces server overhead.

2.3 Consistency of Web Data

A large Web proxy log is analyzed in [13] and it is discovered that 38% of Web server responses contain an impossible (future) Date field and 0.3% contain an impossible Last-Modified value. In a small fraction of cases, returned date fields are completely wrong, off by several hours and even months.

A study of Web caching effectiveness [4] suggests that most proxy caches access a document only once but that they access many different documents on the same server (e.g. a news article and associated graphics files). Also, since updates are mostly concentrated on few, popular documents (e.g. headline news), the authors claim that a server-based invalidation scheme would provide strong consistency with a minimal impact on network bandwidth.

As a first step toward Web Service caching, the Response Cache scheme [17] proposes adding a ResponseCache element to SOAP headers. The header specifies how to compute a unique cache key for a given SOAP message and what the associated expiration time (DeltaFreshness) is. A pair of XPath expression is used in calculating the cache key. A Service Expression determines the Service Key for a service endpoint URI and identifies the particular Web Service being invoked. A Message Expression, determined by the Service Key, is used to compute the Message Key (cache key) for a given Web Service response.

Enabling Web Service caching in Microsoft ASP.NET is described in [18]. Several methods exist, including using explicit HTTP header values, specifying a CacheDuration modifier for the ASP.NET output cache in the programming language, and using an explicit caching API for application caching.

3. REQUIREMENTS FOR PRACTICAL WEB SERVICE CACHING

As described in the Introduction, our focus is on using Web Service protocols for application and business process integration, an example of which is supply chain management. Participants of such a collaboration may be at a great distance of each other, but they also maintain continuous network activity and are closely coupled due to business agreements that are in place. This means that network disruptions have an immediate impact on system availability on both sides of the collaboration. But it also means that coherence protocol messages (invalidations) can be piggybacked to existing data messages (SOAP responses) with little additional cost.

It is expected that, at a minimum, caches support a time-to-live (TTL) protocol because business data may very well have an inherent expiration characteristic due to a periodic business process, for instance end-of-month bill processing. Given the weight of business collaborations, the number of such couplings does not preclude explicit server-based invalidations. Leases provide a natural mechanism to further reduce the number of invalidation callbacks needed.

Invalidation calls are best viewed as Web Services themselves. In other words, the cache should make itself available as a Web Service, thereby making the communication between a cache and server symmetric. This is also required to maintain platform-independence.

Network outages are disastrous to business processes, yet unavoidable. It is imperative that business processes be allowed to continue in degraded mode using stale data, as long as appropriate controls are in place to limit the damage staleness might cause. For some applications, e.g. forecasting and mining, it may be entirely acceptable to use moderately out-of-date data.

```
<soap:Envelope>
  <soap:Body>
    <GetTrackingInfo xmlns="http://fedex.com/">
      <TrackingNumber>...</TrackingNumber>
      <LicenseKey>...</LicenseKey>
    </GetTrackingInfo>
  </soap:Body>
</soap:Envelope>
```

Figure 1: Sample SOAP Request.

Transparency is important—the existence of a cache should not be visible to an application developer, except perhaps via configuration options that let the developer specify data freshness requirements for his or her application. Caching should be done "under the cover" with no special APIs to bother the developer with.

As suggested in the Response Cache proposal [17], flexibility in the coherence protocol is maximized when the Web Service provider is allowed to specify what constitutes a cache key. Since Web Service data is likely to be XML-formatted, the use of XPath expressions to specify cache keys is a natural choice. Similarly, coherence protocol information such as invalidations are conveniently relayed as SOAP header elements. Both decisions also very much supported by the current trend to add native XML storage, indexing, and query capabilities to database engines. As a result, a Web Service cache becomes a large repository of XML messages (documents) accessed via XQuery and XPath expressions.

Piggybacking invalidation information to regular data messages from server to cache involves adding a SOAP header element that specifies the cache keys of expired items. Since the actual item that is expiring is not transmitted in the invalidation message, the cache key cannot be an XPath expression but rather needs to be a literal value.

As with HTTP proxy caches, which do not cache data that is private and requires authentication, Web Services responses containing private data must not be cached.

4. DATABASE SUPPORT FOR WEB SERVICE CACHING

4.1 Database Web Service Consumer

Database systems provide a reliable and scalable solution for integrating and processing data from multiple, heterogeneous data sources. A natural way to extend the reach of a database engine to heterogeneous data sources is to add a generic Web Service consumer function. The consumer function can be invoked through SQL functions to interact with any Web Service provider. Close integration with the database engine permits efficient coupling of database data (relational and XML) with Web Services and appropriate query planning, optimization, parallelization, and execution just like with any other SQL query.

The following query retrieves package tracking information

```
<soap:Envelope>
  <soap:Body>
    <GetTrackingInfoResponse xmlns="http://fedex.com/">
      <GetTrackingInfoResult>
        <TrackingNumber>...</TrackingNumber>
        <ReferenceNumber>...</ReferenceNumber>
        <DeliveredTo>...</DeliveredTo>
        <DeliveryLocation>...</DeliveryLocation>
        <DeliveryDateTime>...</DeliveryDateTime>
        <SignedBy>...</SignedBy>
        ... other data ...
      </GetTrackingInfoResult>
    </GetTrackingInfoResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2: Sample SOAP Response.

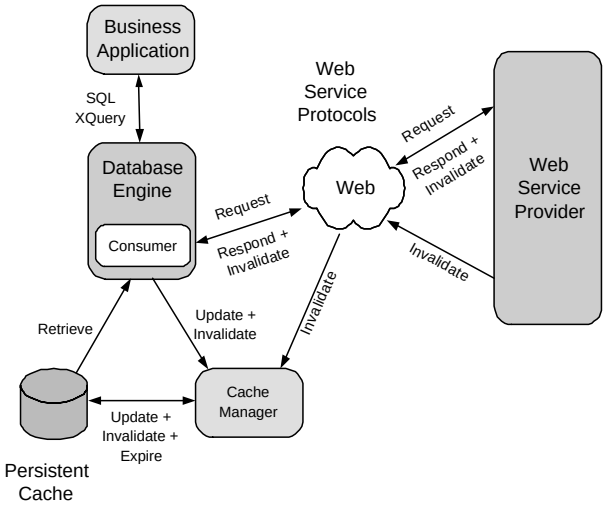

Figure 3: Components of Web Service Caching.

for all overdue orders. The GetTrackingInfo function in the query invokes an external Web Service that returns package tracking information for each package.

```
SELECT orderid, GetTrackingInfo (trackingid)
FROM Order
WHERE order_status = ´overdue´;
```

The consumer function generates a SOAP request envelope and sends it to a service provider (Figure 1). The service provider acts upon the request and sends back a SOAP response envelope (Figure 2).

4.2 Persistent Cache

The architecture of the Web Service caching scheme is illustrated in Figure 3. The cache is stored in database tables whose schema is shown in Figure 4. Four tables are used: current cache (CC), historical cache (HC), service expressions (SE), and message expressions (ME). CC stores the most recent response from a Web Service for a given cache key. HC is a temporal dataset where some number of old

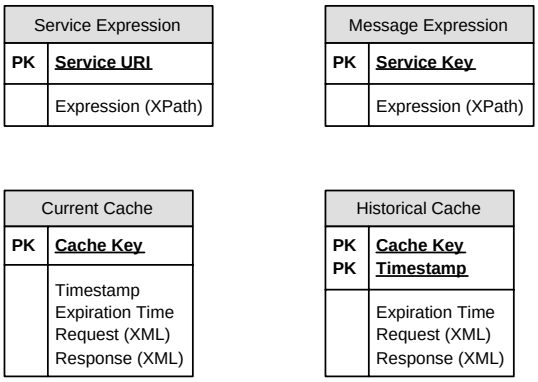

Figure 4: Schema for Persistent Cache.

responses is stored, with associated timestamps. The historical dataset is separated from the current cache so that queries requiring the most recent cache value can be executed efficiently. The three-headed cache lookup queries described in Section 4.4 are an example of this. The SE and ME tables store instructions for how to compute a globally unique key for a given Web Service request and are described in the next section.

Having the database system store Web Service responses in a historical cache enables analysis and mining of that data, which—given that our focus is on business data—is an important reason for the database system to listen to ongoing business processes. We expect the database to provide the scalability needed for large caches and to make it possible to handle multiple versions of data, which enables time travel and monitoring and resolution of problems in business processes (dashboarding). Replication of cache tables across sites enables multiple cooperative caches [10], further increasing the availability and performance of Web Services data.

The cache is populated by a modified Web Service consumer function that forwards received Web Service responses, as well as piggybacked invalidations, to a Cache Manager via a local, high-performance message queue. In order not to add overhead to the original transaction, the actual insertion of a cache record into cache tables occurs in a separate transaction in the Cache Manager.

4.3 Cache Key Computation

The cache key for CC and HC is computed using two XPath expressions as proposed in [17] and illustrated in Figure 5. Briefly, a Service Expression determines the Service Key for a particular service endpoint URI and uniquely identifies the particular Web Service being invoked. Note that the service endpoint URI itself is not a globally unique Service Key, as each service endpoint URI may provide multiple Web Services. The Service Expression is stored in the SE table and is not likely to change frequently.

The Service Key is used to lookup a corresponding Message Expression in the ME table. The Message Expression is used to compute a Message Key for a given Web Service request.

```
<ResponseCache>
  <serviceKey>
    concat(namespace-uri(/*/*[local-name()='Body']/*),
           local-name(/*/*[local-name()='Body']/*))
  </serviceKey>
  <messageKey>//TrackingNumber/text()</messageKey>
  <coherence>
    <delta-freshness>300</delta-freshness>
  </coherence>
</ResponseCache>
```

Figure 5: Sample Service and Message Keys.

A globally unique cache key is obtained by concatenating a Service Key and a Message Key.

As an example, consider the expressions one might use for a package tracking Web Service (Figure 5). The Service Expression specifies that the Service Key consists of the namespace and name of the XML element immediately below the SOAP Body element. For RPC-style requests, this is the name of the SOAP method being called. For a package tracking service, the expression might evaluate to `http://-fedex.com/GetTrackingInfo` and is a reasonable definition of a Service Key for that service. The Message Key is the value of the text node immediately below the "TrackingNumber" element and contains the package tracking number, for instance 285982392432. Combining the Service Key with the Message Key, we get a globally unique cache key `http://fedex.com/GetTrackingInfo285982392432`.

A range of alternatives exist for computing a cache key based on the SOAP request envelope. The simplest option is to treat the entire request as a cache key. This is overly conservative, however, as minor variations in XML syntax, such as extra whitespace or a different order of attributes, would produce a different cache key. This problem is solved by standardizing the format of request envelopes via XML canonicalization [19]. However, the cache key is still overly conservative because not every part of the request is important for guaranteeing the uniqueness of the key. For example, the package tracking service takes an additional parameter "LicenseKey" which authorizes the consumer to invoke the service. The response does not functionally depend on the license key, and, hence, the license key parameter should not be part of the cache key. The suggested method is therefore to ask the service provider to tell the consumer (via the Service Expression and Message Expression) what part of the request *does* functionally determine the response.

4.4 Query Processing

Invocation and rewriting of Web Service queries is illustrated in Figure 6. A database application invokes a Web Service consumer function defined in the database, providing the necessary service parameters such as service provider URI, method name, and method arguments. For document-style services, a service provider URI is complemented by an XML document. The database engine rewrites the function invocation as three subqueries tied together by a "switch." The switch employs the semantics of a COALESCE function and attempts to invoke the subqueries in order until one succeeds (returns a non-null value). The value of the successful subquery is then returned as the value of the switch.

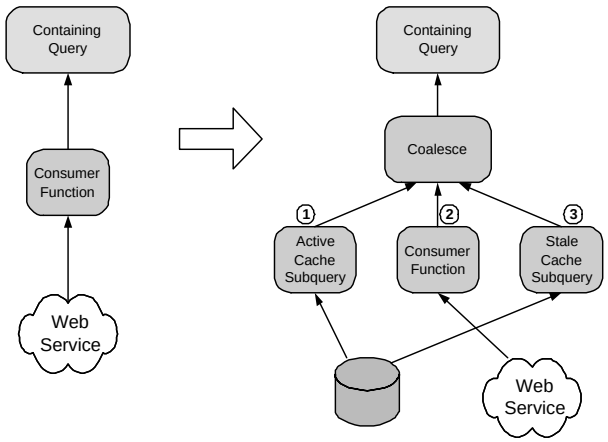

Figure 6: Rewriting of Three-Headed Query.

The first subquery queries the current cache table CC and retrieves active (non-expired) data. If no match is found in the cache, the Web Service is invoked. If the Web Service fails due to a network outage or other reason, a third subquery is invoked to get stale data from the cache. Note that query rewriting is analogous to URL rewriting and DNS redirection in Content Delivery Networks (CDN) [15].

Once the query is rewritten to take advantage of the cache, it can be further improved to exploit query parallelism, which is a natural fit for deployments in Grid environments.

4.5 Cache Manager

The Cache Manager component of the system implements the cache coherence logic, as illustrated in Figure 7. The Cache Manager receives a cache record or invalidation message (command) from the database engine through a local message queue. An invalidation message may also be received from the Web Service provider directly, in case it has no regular data messages to piggyback the invalidation message onto. The Cache Manager inserts/updates the current cache table CC with the cache record or removes cache records based on invalidation information. If a burst of cache records is received, the records are inserted/updated in batches, not one-by-one. Cache records are also inserted to the historical cache table HC if the Cache Manager is configured to keep multiple versions of Web Service responses. Extraneous copies are removed from the historical cache table HC at regular intervals during inserts/updates.

The Cache Manager can also receive meta-level commands such as Pause, Continue, and Set Variable (e.g. insert batch size). At regular intervals during idle periods, the Cache Manager deletes expired cache records if configured not to keep stale data. Automatic refresh/prefetch causes the Cache Manager to simply re-invoke the original Web Service request with the parameters stored in the cache record. The Web Service consumer function of the database retrieves the new data and forwards the data back to the Cache Manager via its message queue.

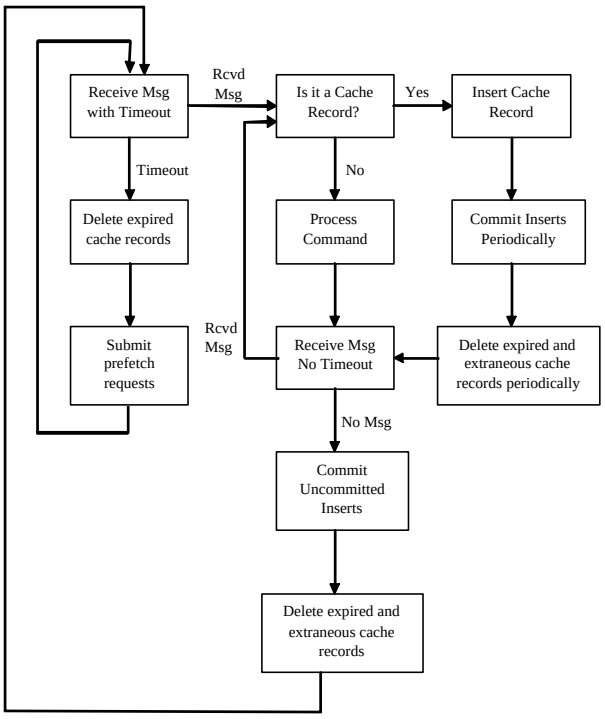

Figure 7: Cache Manager Event Processing.

Allowing explicit invalidation callbacks from the Web Service provider to the Cache Manager requires that the Cache Manager expose its queue as a Web Service. A small Web Service manager (Web server plus servlet) is required to handle these incoming cache invalidations.

4.6 Prototype Implementation

A prototype of the query processing logic described in this paper has been implemented in the query compiler of DB2. Query rewriting is done by directly modifying the Query Graph Model (QGM), the internal representation of queries in DB2. The modification is done by adding a new rewrite rule to an early phase in query compilation. This permits subsequent planning and optimization in later phases of query compilation.

The prototype exploits the Web Service consumer function provided as part of DB2 V8.1 [7]. In addition to returning a Web Service response to the query runtime component, the consumer function is modified to also route the response to the Cache Manager via IBM MQSeries message queues. The Cache Manager is an external DB2 agent and is exposed as a Web Service for invalidation callbacks via DB2 Web Object Runtime Framework (WORF).

5. PERFORMANCE STUDY

A series of performance tests were conducted using the Web Service consumer function in both cached and non-cached configurations. Our goal was three-fold: confirm that adding the cache lookup logic did not increase the cost of a non-cached access (cache miss), measure the cost of a successful cache retrieval (cache hit), and measure the speed-up achieved in Web Service calls as a function of the degree of parallelism.

```
values db2xml.soaphttpc (
  ´http://localhost/TestServices/servlet/rpcrouter´,
  ´´,
  xml2clob (
    XMLElement (
      NAME "ns:getTime",
      XMLAttributes (´http://tempuri.org/GetTime´
                     as "xmlns:ns" )
    )
  )
)
```

Figure 8: Sample Web Service invocation.

Four Web Services were exercised in the study (Table 1). The services differed in terms of the location of the server relative to the consumer: `GetTime` and `GetSleep` invoked a local Web Service running on the same machine as the consumer, while `GetProc` invoked a service on another server on the local network. `GetTemp` invoked a service hosted on the Internet at `servicemethods.com`.

Four cache configurations were tested (Table 2). No Cache refers to the non-caching consumer function shipping in DB2 V8.1, while the other three configurations are prototypes and provide improvements over the product version. Disabled Cache adds caching logic to the consumer function but the feature is disabled (a runtime flag is used to check if the feature is enabled or not). Cache Miss performs a cache lookup using SQL and DB2 CLI but is configured not to find valid data in the cache. Instead, it invokes the Web Service and then inserts the response into the cache. Cache Hit performs a cache lookup, finds valid data in the cache and returns it. All four Web Services were run using all four cache configurations. In addition, the services were run in the No Cache configuration with different degrees of parallelism to measure their speed-up.

The consumer function was invoked in DB2 V8.1 running on Windows 2000 with a 933 MHz Pentium 4 CPU and 512 MB of memory. Web Service calls were initiated by entering a SQL statement in DB2 that constructed the request envelope in XML using SQL/XML publishing functions and then invoked the Web Service consumer function (Figure 8). `GetTime` and `GetSleep` were implemented in Java and hosted in the WebSphere Studio Application Developer (WSAD) V5.0 environment running on the same machine as DB2. `GetProc` was implemented in Java and hosted in WSAD V5.0 on a 2.2 GHz Pentium 4 machine with 512 MB of memory and running Windows 2000. The implementation details and hardware configuration of `GetTemp` are unknown to the authors.

Figure 9 shows the overhead added by the caching logic in a cache miss situation and the benefit of caching in a cache hit. For each of the four Web Services, the chart shows the elapsed time of different cache configurations relative to the cost of a non-cached invocation. Therefore, for all services the cost of No Cache is 1. The cost of Disabled Cache and Cache Miss was expected to be close to 1 and perhaps even a

Table 1: Web Services exercised in the study.

Web Service	Description	Typical response time
`GetTime`	Get current time from local machine	10 ms
`GetProc`	Get workflow process data from server on LAN	200 ms
`GetTemp`	Get weather information from server on Internet	400 ms
`GetSleep`	Do nothing for 1 second on server on LAN	1000 ms

Table 2: Web Service cache configurations compared in the study.

Name	Description	Code status
No Cache	Non-caching consumer function	Ships in DB2 V8.1
Disabled Cache	Caching function with cache feature disabled	Prototype
Cache Miss	Cache lookup (miss) + Web Service invocation + cache insert	Prototype
Cache Hit	Cache lookup (hit)	Prototype

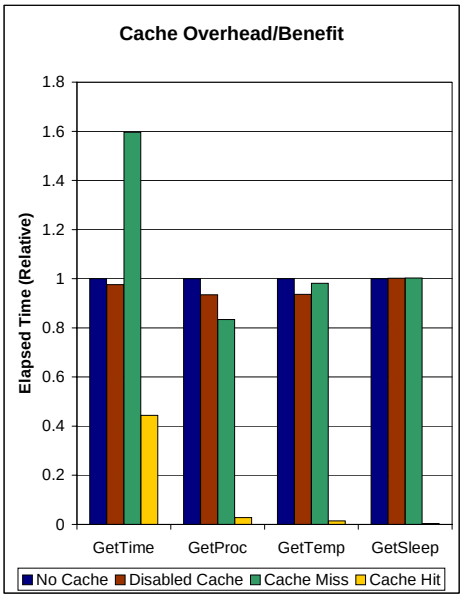

Figure 9: Cache overhead/benefit of Web Service invocations

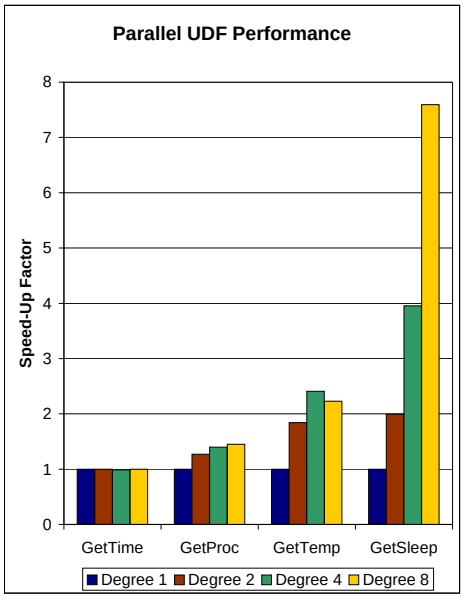

Figure 10: Parallelizability of Web Service invocations

bit lower because of minor performance improvements (not related to caching) made to the prototype code over the shipping product code.

Indeed, both Disabled Cache and Cache Miss ran slightly faster than No Cache, except for the `GetTime` service where Cache Miss was about 60% more expensive than No Cache. Recall that in absolute terms `GetTime` is the fastest of the Web Services, having a typical response time of only 10 ms, therefore an extra cost of a few milliseconds in cache lookup time shows up as a dramatic increase in relative terms.

Cache Hit cost appears very small compared to all other configurations. `GetTime` is an exception again due to its short overall elapsed time. Instead of a 10 ms Web Service response time, the consumer function spends about 40% or 4 ms in performing a cache lookup and returning the result. The 4 ms cache lookup cost translates to small or negligible relative costs for the more expensive Web Services.

Because Web Services are expected to involve lengthy network latencies and overall service times (perhaps ranging to hours or days), it is important to be able to parallelize these calls. Figure 10 shows the speed-up of the four Web Ser-

vices as a function of N, the degree of parallelism. `GetTime` involves very little CPU and is mostly network bound. Increasing N yields no performance benefits, as the requests merely queue up either in the networking layer or at the web service provider.

A small increase in speed-up is seen with `GetProc`. The difference between $N = 4$ and $N = 8$ is very small and indicates that the network and/or CPU at the service provider is saturating. For `GetTemp`, the initial speed-up from $N = 1$ to $N = 2$ is significant, almost linear, but the improvement levels off after $N = 2$, indicating saturation.

As an extreme example of parallelizability, `GetSleep` shows almost linear speed-up with N. The sleep function requires little or no CPU cycles and many callers can be supported without saturation. At $N = 8$, the speed-up appears to level off, indicating the onset of network contention.

6. CACHE INVALIDATION

When communicating with a Web Service provider (*server*) that is capable of providing invalidation callbacks, the database (*client*) needs to inform the server that its response was cached and must be invalidated when the server data changes. The server records these *invalidation subscriptions* in a list and makes invalidation callbacks if and when the data does change.

A subscription has an associated expiration time, typically the same value that was provided by the server in the first place to indicate the freshness of the data. When the expiration time passes, the server knows that the client no longer has a valid copy of the data, and therefore no longer requires invalidation callbacks. This scheme is very similar to the concept of leases except that strict leases give the caching client exclusive write access to the data. Our database cache is read only so exclusive access remains with the server.

As mentioned previously, invalidation callbacks are best implemented using Web Services for reasons of symmetry. In order for the server to make such Web Service callbacks, it needs to know the Web Service information (subscription) of the client. The client provides its service endpoint, SOAP action, and message to be delivered in a SOAP header element `InvalidationSubscription`, as illustrated in Figure 11. As with invalidation callbacks, the information is piggybacked to existing data messages and incurs only a small incremental cost.

When the server wants to invalidate a cached data item, it piggybacks an *invalidation request* to an outgoing data message whose destination is the same as that of the invalidation request. The content of the invalidation request is the `message` element. Figure 12 illustrates the invalidation request sent to service endpoint `http://host.com/servlet/rpcrouter` with SOAP action `invalidate`. If no data message is scheduled to go out, the server will invoke the invalidation Web Service of the client explicitly. The name of the method to invoke and all its parameters are contained in the `message` element (Figure 11).

```
<InvalidationSubscription>
  <expires>2003-03-15-09.00.30</expires>
  <serviceURI>
    http://host.com/servlet/rpcrouter
  </serviceURI>
  <action>invalidate</action>
  <message>
    <ns:invalidate xmlns:ns="http://tempuri.org/">
      <serviceKey>...</serviceKey>
      <messageKey>...</messageKey>
    </ns:invalidate>
  </message>
</InvalidationSubscription>
```

Figure 11: Cache Invalidation Subscription.

```
<InvalidationRequest>
  <ns:invalidate xmlns:ns="http://tempuri.org/">
    <serviceKey>...</serviceKey>
    <messageKey>...</messageKey>
  </ns:invalidate>
</InvalidationRequest>
```

Figure 12: Cache Invalidation Request.

7. CONCLUSION

We have discussed the role of Web Service protocols in enabling businesses to link applications together within and across enterprise networks. The extensibility of the protocols and the use of XML as a high-level data format make integration possible, but they also create formidable challenges in terms of application performance and availability.

We have described an approach to mitigating performance and availability problems by caching responses from Web Service requests. We discussed the requirements for Web Service caching and outlined a plan for incorporating support for it into a database engine. Existing research on cache coherence models (e.g. expiration times and invalidation callbacks) and cooperative caches apply directly to Web Service caching. Since Web Service protocols build on existing Web protocols, such as HTTP, it is attractive to reuse as much of existing Web caching techniques as possible. We note, however, that Web Service protocol messages are usually POSTed and the body of posted messages contains key information (e.g. SOAP method name and arguments) which HTTP proxy caches are not able to exploit. Therefore, existing HTTP proxy caching mechanisms are not adequate for Web Service caching.

Our scheme involves a persistent database cache of Web Service responses, which gets utilized in Web Service queries automatically rewritten by the database. The rewritten query first looks for active (non-expired) data in the cache before attempting to invoke the Web Service. Should the invocation fail due to a network or other reason, another lookup to the cache to find stale data is performed. We note that allowing for a limited degree of staleness is important to increase the availability of the database and any applications built on top of it. A Cache Manager maintains the persistent cache and removes expired data when necessary or when explicitly requested by a Web Service provider via invalidation

callbacks.

Our future work focuses on further improvements to the caching protocol in support of business process integration. Policies also need to be developed for declaring what to keep in the cache and what can be removed. These policies may be affected by business processes that are explicitly declared using Business Process Execution Language (BPEL) and other specifications. Our plan is also to conduct extensive performance experiments, validating the efficacy of the caching and invalidation scheme with real business processes.

8. REFERENCES

[1] Randal C. Burns, Robert M. Rees, and Darrell D. E. Long. An analytical study of opportunistic lease renewal. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS)*, Phoenix, Arizona, April 2001.

[2] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[3] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant J. Shenoy. Adaptive push-pull: Disseminating dynamic Web data. In *Proceedings of the Tenth International World Wide Web Conference*, pages 265–274, Hong Kong, May 2001.

[4] Felicia Doswell and Marc Abrams. The effectiveness of cache coherence implemented on the Web. Technical Report TR-00-02, Virginia Polytechnic Institute, Computer Science, May 2000.

[5] Carl G. Gray and David R. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating System Principles (SOSP)*, pages 202–210, Litchfield Park, Arizona, December 1989.

[6] James Gwertzman and Margo I. Seltzer. World Wide Web cache consistency. In *USENIX Annual Technical Conference*, pages 141–152, January 1996.

[7] IBM. DB2 Developer Domain - Web Services, October 2002. http://www7b.software.ibm.com/dmdd/zones/-webservices/.

[8] Cristian Ionitoiu and Maria Angi. An adaptive caching and replication mechanism for WWW. In *Proceedings of Time and the Web Seminar, Staffordshire University*, June 1997.

[9] Susan Malaika, Constance J. Nelin, Rong Qu, Berthold Reinwald, and Daniel C. Wolfson. DB2 and Web services. *IBM Systems Journal*, 41(4):666–685, 2002.

[10] Radhika Malpani, Jacob Lorch, and David Berger. Making World Wide Web caching servers cooperate. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[11] Kuassi Mensah and Ekkehard Rohwedder. Database web services, November 2002. An Oracle White Paper.

[12] Microsoft. SQLXML 3.0, 2002.

[13] Jeffrey C. Mogul. Errors in timestamp-based HTTP header values. Technical Report 99/3, Compaq Computer Corporation Western Research Laboratory (WRL), December 1999.

[14] C. Mohan. Tutorial: Caching technologies for Web applications. Hong Kong, August 2002. Tutorial at the International Conference on Very Large Data Bases (VLDB).

[15] Anoop Ninan, Purushottam Kulkarni, Prashant Shenoy, Krithi Ramamritham, and Renu Tewari. Cooperative leases: Scalable consistency maintenance in content distribution networks. In *Proceedings of the Eleventh International World Wide Web Conference*, Honolulu, Hawaii, May 2002.

[16] W3C Note. Simple object access protocol (SOAP) 1.1, May 2000. http://www.w3.org/TR/SOAP/.

[17] Mark Nottingham. SOAP optimisation modules: Response caching, August 2001. http://lists.w3.org/Archives/Public/www-ws/-2001Aug/att-0000/01-ResponseCache.html.

[18] Matt Powell. XML Web service caching strategies. Technical Report 04/17/2002, Microsoft Developer Network Library (MSDN), April 2002.

[19] W3C Recommendation. Canonical XML, March 2001. http://www.w3.org/TR/xml-c14n.

[20] Devavrat Shah, Sundar Iyer, Balaji Prabhakar, and Nick McKeown. Maintaining coherency of dynamic data in cooperating repositories. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, August 2002.

[21] Raghav Srinivasan, Chao Liang, and Krithi Ramamritham. Maintaining temporal coherency of virtual warehouses. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS)*, Madrid, Spain, December 1998.

[22] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Design considerations for distributed caching on the internet. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 273–284, Austin, Texas, June 1999.

[23] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, April 1997.