The promise of software as a service is becoming a reality with many ASPs (application service providers). Organizations using ASPs and third-party vendors that provide value-added products to ASPs need to integrate with them. ASPs enable this integration by providing Web service-based APIs. There are significant differences between integrating with ASPs over the Internet and integrating with a local application. When integrating with ASPs, users have to consider a number of issues, including latency, unavailability, upgrades, performance, load limiting, and lack of transaction support.

WEB SERVICE API

The integration APIs provided by ASPs are usually based on Web services. The ASPs provide documentation and WSDL (Web Services Description Language) files for the APIs. The user of the API uses the WSDL files to generate the Web service client using a Web service toolkit (e.g., Apache Axis). An ASP may provide a feature in its software to create custom fields for various entities. For example, CRM (customer relationship management) systems are likely to provide custom fields on such entities as customer accounts, since organizations will have specific information they want to track about a customer. If an ASP supports custom fields, it will usually provide two separate WSDL files: one generated specifically for an organization based on the custom fields defined by that organization, and one that is generic.

The WSDL generated specifically for an organization allows the organization to access custom fields in the same way as the standard fields, which allows the developer to be unaware of whether a given field is cus-

tom or standard. The generic WSDL provides a generic way to access the custom fields and would be used by a third-party vendor that is not developing its product for a specific customer and hence does not need any specific sets of custom fields. Even an organization developing an integration to its own system may be better off using the generic version, especially if it anticipates adding more custom fields and does not want to keep retrieving the organization-specific WSDL and regenerating its Web services client.

This article is based on experience with three ASPs:

- Salesforce.com, a hosted CRM product¹
- CRM OnDemand.com by Siebel, also a hosted CRM product²
- QuickBooks Online Edition by Intuit, a hosted accounting product³

SERVICE PROVISION ISSUES

With most ASPs, only one version of their software is in production at any given time. This means that all customers will be upgraded to the latest version when the ASP releases it. The ASP makes the new version of its software available in a pre-release environment a few months ahead of time so that integrated applications can test against the new release before it goes into production.

ASPs provide a different URL for each version of their APIs. Theoretically, the existing versions of APIs should not be affected by a new release of the software; however, there can be unintended changes in behavior as a result of changes made in the implementation. For example, Salesforce.com had a line-item entity in which you could specify any two of the fields—price, quantity,

LEN TAKEUCHI, SALESCENTRIX

ASPs

**The Integration
Challenge**

ASPs

The Integration Challenge

and amount—and the third field was calculated (price x quantity = amount). Up to a particular upgrade release, you could either not specify a value for the field you were having calculated or set that field to null. Once the upgrade occurred, setting the field to null resulted in an error being reported, which broke our integration.

It is imperative, therefore, that a full regression test be performed against the new release during the pre-release period and problems reported to the ASP and/or changes made to ensure that the integration continues to work after the new release. ASPs periodically release new versions of their APIs, often coinciding with a new release of their software, but they will continue to support older versions of their APIs for some time.

ASPs may or may not provide SLAs (service-level agreements) on their availability and performance. Some will provide SLAs only for their biggest customers, but they will always provide prior notice for scheduled downtimes. Some ASPs partition their customers based on location (e.g., Americas, Asia, Europe) and allocate each set to a different server cluster. This allows them to schedule downtimes outside of most of their customers' business hours. As much as these ASPs try to avoid downtime by having redundancy in their hosting infrastructure, there will inevitably be some unscheduled downtime that may last a few hours or even days. Therefore, the application integrating with the ASP needs to be designed to provide as much of its functionality as possible even if an ASP is not available. This may require that some information be duplicated between systems.

An ASP's efforts to maximize availability include having redundancy in its ISP source. The Internet address of the service provider may change if the ASP goes to its backup ISP or if it makes changes to its hosting environment (e.g., moving to a different data center). As a result, an application integrating with an ASP should not cache results of DNS lookups on the ASP's access points beyond its TTL (time to live) values. Java's default behavior for caching a successful DNS lookup is to cache forever. If

your application or Web server is a Java application, the caching period will have to be adjusted to something more appropriate to ensure that the application continues to work after an ASP's IP address changes.

ASPs have only one version of their software in production at any given time, but they may have different editions of the software with slightly different feature sets. This means that not all features of an API are available for all customers. The application integrating with an ASP will either have to get the edition information directly (if available through the API) or ascertain what features are enabled and behave appropriately.

The shared tenancy model of ASPs makes it imperative for them to protect themselves against a particular organization using too much of their resources. In the case of API usage, they protect themselves by placing limits on:

- Number of API login sessions
- Rate at which requests are submitted (at the organization level or per API login session, depending on the provider)
- Request size
- Query size
- Number of instances for calls where the interface provides for more than one instance to be created/modified/deleted in a single call
- Number of instances returned from a query
- Execution time of a request in general or queries in particular

How the application handles these limitations depends on whether the operation being performed involves an interactive user (attended operation), or the application is performing a background task (unattended operation). When an attended operation exceeds session limitations and request rate limitations, the application can tell the user to try again later. If an unattended operation exceeds the limitations, then either the error needs to be reported using some sort of notification mechanism, or a retry mechanism will be required.

Request size, query size, and limitations on the number of instances input to a call or returned from a call are published in the API documentation. It is up to developers to be aware of these limitations and design their code so that they are not exceeded. For example, if a developer is building a request based on a data set returned from a previous query, the developer must be aware of the size limit and design the code so that the data set is partitioned into several requests; otherwise, the built-up request could exceed the limitations. Developers must also be aware of execution time limits and write their code in such a way as not to exceed them—for

example, by reducing the complexity of a query. During the development and QA cycle, sufficient testing has to be done with large data sets to ensure that execution time limits are not exceeded in a real-world situation. This testing can prevent having to patch the application after its release.

SECURITY

Security in invoking APIs over the Internet is achieved through the use of HTTPS (secure HTTP). Authentication is typically done through a login call where credentials (user ID and password) are passed in. The login establishes a session in which API calls can be made. The session will expire after a certain amount of time following login or after a certain amount of time of inactivity, depending on the ASP. The integrated application will either have to keep the connection alive (which is possible only if timeout is based on inactivity) or re-login preemptively before expiry or detect expiry after the fact and re-login.

The use of two-way authentication with a client certificate is fairly uncommon. Salesforce.com and CRM OnDemand use user ID and password as credentials. QuickBooks Online Edition requires the use of a client certificate if the application accessing the API is itself a server application (Web-based) but not if it is a desktop application. This distinction is probably a concession to the fact that the use of client certificates is fairly uncommon for desktop applications.

Security standards such as WS-Security do not seem to be high on the priority list of the specific ASPs discussed in this article. This is likely because most users of their APIs probably call them directly; hence, transport-level security (SSL) is sufficient. In such a case there is no need for message-level security, since no intermediaries are involved.

Single sign-on, on the other hand, is high on the ASPs' priority lists. For example, Salesforce.com supports a delegated authentication protocol where it makes a secure Web services call to an endpoint that is defined by the customer to provide a flexible mechanism for supporting single sign-on.

Some ASPs provide a way to propagate an interactive login session to an application. Salesforce.com supports sharing of login sessions between an interactive session at its Web site and API access by a Web application. This is done through a Web link feature that allows for configuring Salesforce.com to display links that, when invoked, will result in the Web browser going to the URL configured for the Web link. It is possible to configure a Web

link to pass session ID as a parameter when the browser is redirected to the URL, so that when a Web application receives the redirect to the URL, it can call back to Salesforce.com through its API using the session ID (no need to login again through the API).

RELIABILITY

There are reliability issues in integrating with ASPs using Web service-based APIs over the Internet: failures caused by unreliable communications, ASP unavailabilities, and resource usage limitations imposed by ASPs.

The applications integrating with ASPs have to be able to handle these failures. If the operation being performed is attended, then the application should convey an appropriate error message to the user. If the application is performing an unattended operation, then the application needs to have a notification mechanism for reporting failures or a retry mechanism to ensure reliability. It is important for the application to differentiate error handling based on the failure being reported. For example, there is no point in retrying if a query execution time limit is being exceeded since the error will occur

every time for the particular request until the application is updated to reformulate the query.

Communication problems such as being disconnected or timing out while waiting for a response from the ASP are particularly problematic since the application does not know for sure whether the operation has completed successfully. It would be helpful if the APIs offered some means of providing a caller-supplied request ID, subsequently allowing the caller to query for the result using the ID, but this type of mechanism is not usually provided where the API is synchronous. If the request is idempotent, then the application can retry the request when the results are unknown. Most modification operations in these APIs tend to be non-idempotent, so in these cases the application will have to do its best to determine after the fact whether the request has been successfully processed.

One approach to determine after the fact whether an

ASPs

The Integration Challenge

update request has been successfully processed is to record a unique request ID in a custom field in the entity during the update so that the entity can be subsequently queried and this field inspected to see if the modification has in fact been performed. Since updates to this entity and hence the field holding the request ID may be done by others before a query can be issued, the request ID would have to be appended to this field rather than overwriting it. If we simply read the field, append the request ID to the field value, and then do an update, we may wind up wiping out a request ID recorded by someone else if it was done between the time we read the field and the time we wrote back the field. Clearly, this scheme works reliably only if we can lock the entity between the time we read the field and when we write it back. With this scheme, the request IDs will build up (as they are appended), so there has to be some strategy for getting rid of the earlier request IDs.

There are some emerging standards for transaction support for Web services,⁵ but the ASPs discussed in this article do not support them. In fact, they do not provide any support for transaction processing or locking at all, and such support is not likely in the near future. This means that there is no provision for transaction demarcation and committing or rolling back transactions. If an application has to do a set of updates in an all-or-nothing fashion, the application can attempt to undo successful updates or redo unsuccessful updates if the initial attempt at the set of updates was only partially successful. These approaches will not always work as the subsequent compensating actions may also fail. Also, because each change within the set is made public immediately, others see intermediate states even if the compensating actions are successful. Note that intermediate states are public (though for short periods of time) even when all the operations in the set are successful.

To take a concrete example, one of the main entities in a CRM system is an *opportunity*. An opportunity is a potential sale to a customer and can have line items

associated with it. These line items record the items and quantities that the customer is interested in. Opportunity and its line items are similar in nature to a *sales order* and its line items. In Salesforce.com and CRM OnDemand, the opportunity and its line items are not updated in a single call in the API. In fact, the updating of each line item is a separate update. If a change is made to an opportunity such that the quantity from the first line item is increased by 1 and the quantity of the second line item is reduced by 1 but the update to the second line fails, then we wind up with a partially updated opportunity. With lack of transaction support, the best strategy may be simply to retry the update on the second line and if it fails again, try to roll back the change to the first line. If neither of these is successful, then the error will have to be reported and left up to the user to correct the situation. Retrying at a later time is not necessarily a good option in this case since others may update the same opportunity before the retry occurs; we don't want the retry to wipe out these changes.

The behavior of the interactive application provided by the ASP can cause problems with the application seeing partially updated states. For example, with Salesforce.com and CRM OnDemand, the line items associated with an opportunity are updated by the user one at a time. If the user is in the middle of making a set of changes to the line items and the application being integrated happens to query for the opportunity's line items, then the application will see a partially updated state (i.e., updates to line items have not been completed). The application will have to come up with some strategies to minimize the chances of seeing these intermediate states.

One approach the application can take is to check the modified times of the line items returned in the query. If they are recent (within a certain threshold), then the application can wait and try again later. This does not necessarily guarantee that the user has completed the set of changes since there is no set time frame for the user to make the changes, but this approach will lessen the chance of seeing intermediate states. If this scheme is used in a system where Opportunities are being periodically pulled from the CRM system, then an intermediate state may be seen in one polling cycle but be corrected in the next cycle. If an intermediate state must not be seen, then a scheme has to be put in place where the user explicitly demarcates the start and end of an editing session. One primitive way to do this is to record in a custom field whether an opportunity is being edited and have the user manually update this field value before and after editing the opportunity.

PERFORMANCE

The main problem with performance when using the APIs provided by ASPs is the per-call overhead. The overhead comes from latency over the Internet, transmission time, marshalling (XML/SOAP) requests on the client side and responses on the ASP side, un-marshalling (XML/SOAP) requests on the ASP side and responses on the client side, and authentication/authorization checking and other overhead at the ASP.

In accessing Salesforce.com and CRM OnDemand (both of these ASPs are located in North America, as are we), we have observed round-trip latency of 25 to 50 milliseconds. To get an estimate of the amount of per-call overhead, we did some tests with Salesforce.com by issuing an API call that returns the server time in GMT (Greenwich Mean Time). The fastest time that we got for completing this call was 150 milliseconds. Because the request and response are very simple (small) for this particular call, the transmission time and marshalling and un-marshalling times are very small contributors to the total call time. Furthermore, the actual processing time should also be very small since this is returning the server time in GMT (and hence no time-zone conversions are likely being done). This means that the per-call overhead is as much as 150 milliseconds including latency. So for Salesforce.com 150 milliseconds is the absolute floor for any call—any call will take 150 milliseconds plus transmission time, marshalling and un-marshalling time, and actual processing time for the logic of the request.

If the ASP supports compression (Salesforce.com does) and compression is supported by the client environment, then it should be used to minimize transmission time. Enabling compression is usually done by setting the AcceptEncoding and ContentEncoding HTTP header fields to the compression format.

Given the large per-call overhead, an application will perform poorly if many separate requests are sent to the ASP. The application needs to do all that it can to minimize the number of separate calls by caching data where appropriate, using mechanisms provided by APIs to send a set of modifications together, retrieving information in groups rather than individually, and avoiding unnecessary (redundant) requests.

The strategies for improving performance when using an API over the Internet are not neces-

sarily novel, but they are particularly important in an environment where per-call overhead is very high.

It is important to cache data as much as possible. The application should have frameworks to cache data so that the same information is not retrieved more than once within the context of an operation. For slow-changing setup data, caching should also be considered between operations. To this end, monitoring the calls made to the ASPs in a running system is useful for locating where redundant retrievals are being performed. The ASPs covered here do not provide for any cache invalidation protocols. The only support for caching is the ability to retrieve objects based on their modified times. Any caching scheme will have to use modified time-based queries at strategic times to refresh the cached data.

The APIs will provide some means of grouping or batching operations within a single call to the ASP. (Note, however, that in the case of the ASPs covered in this article, there are no transaction semantics to a batch.) It is important to take advantage of grouping/batching, especially in cases where the application is itself processing a batch-type operation. An application should not do so by processing each item in the batch individually, as this will almost always result in missing out on opportunities to group operations sent to the ASP and to retrieve information from the ASP in groups rather than individually.

An example of how grouping requests into one call saves a significant amount of time in a batch-type operation is an operation that pushes products up to the CRM system. This operation receives the product information from the source in batches and creates the products in the CRM system. If a separate call to the CRM system is made for each product in the batch, then we will incur the per-call overhead for each product. We found tremendous savings when we sent the group of products in the batch to the CRM system in a single call, thus mitigating the large per-call overhead. The results of creating products in Salesforce.com in one call for different numbers of products are depicted in table 1.

It is important to take advantage of whatever features

TABLE 1

Product Creation Times for
Different Group Sizes

Number of Products	Time	Time for Separate Requests	% Savings in Time for Grouping
1	243 ms	243 ms	-
5	445 ms	1,215 ms	64%
10	689 ms	2,430 ms	72%
20	1,284 ms	4,860 ms	74%

ASPs

The Integration Challenge

the APIs provide to minimize requests sent to the server. One example of such a feature is Salesforce.com's **upsert** command, in which an insert will be done if the object does not exist but an update will be done if the object does exist. This saves having to perform a query first to determine whether an update or insert needs to be done.

The application should try to determine, where possible, whether modifications being made are redundant. If the application happens to have fetched the current contents of an object, it should compare this state to the new state of the object and make sure there are changes before sending a modification request to the server.

As with any API, the performance of operations within the API is going to differ. It is important to identify slow

establish a connection with the ASP. Applications should set response timeouts that are large enough for the operation being sent to the ASP to avoid the Web service client prematurely giving up.

Because these APIs are Web service-based, the performance of the XML parser used by the application, specifically the Web service client, is particularly important. In our case, the client is itself a Web application. When we changed Web server versions, we had issues with XML parser memory usage with the default XML parser provided by the Web server. We changed to using a parser specifically for our Web application, and we chose a new version of the parser we had been using with the old Web server. The memory utilization characteristics improved, but we had performance (CPU utilization) problems. We wound up going back to the XML parser that was in use when we were using the older version of the Web server.

CONCLUSION

Developing an application that integrates with ASPs using their APIs over the Internet requires the use of strategies and techniques to overcome certain obstacles. The application must deal with a communication medium that is not reliable and has high latency, ASPs that are not always available, and APIs that do not provide transaction support. It's a challenge, but the promise of software as a service may make the effort worthwhile. ☐

REFERENCES

1. Salesforce AppExchange API—AppExchange Web Services Developer Guide; <http://www.sforce.com/us/docs/sforce70/wwhelp/wwhimpl/js/html/wwhelp.htm>.
2. Siebel Web Services CRM OnDemand Guide, Version 5.
3. QuickBooks SDK Programmer's Guide, Version 5.0.
4. WS-Security Standard; http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.
5. Web Service Transaction Specifications; <http://www-128.ibm.com/developerworks/library/specification/ws-tx/>.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

LEN TAKEUCHI is a senior software engineer at Salescentrix, where he is involved in the design and development of AccountDynamics, a hosted service that links on-demand CRM solutions to on-demand and premise-based accounting solutions. He holds a M.A.Sc in computer engineering from University of British Columbia.

© 2006 ACM 1542-7730/06/0600 \$5.00

The main problem with performance when using these APIs is the per-call overhead.

operations and use them sparingly. It is important to report slow operations to the ASPs as they will often try to improve the performance of these operations or try to mitigate the impact by introducing alternative forms of the operation. For example, an ASP might group operations of a particular type, at least to minimize the per-call overhead contribution.

It is important for an application to set the various timeout values on the calls being made to an ASP. The setting of these timeouts may be at the communication level or at the Web service client level. For an interactive operation, the application should set the connection establishment timeout to a fairly small value; the user should not have to endure a prolonged wait before the application declares a failure because it is unable to