USING QUERY LANGUAGES AND MOBILE CODE TO REDUCE SERVICE INVOCATION COSTS

R. Szymanski*, N. Palmer
AMSRD-CER-C2-BC
Fort Monmouth, New Jersey 07703

T. Chase
Dept. of Chemistry, Medical Technology and Physics
Monmouth University
West Long Branch, New Jersey 07733

ABSTRACT

This paper discusses an approach to reducing the overhead associated with service invocation while at the same time increasing the usefulness of the data returned by services. The described technique is based on shifting computation from the client side to the service side thereby reducing the number of calls the client must make to the service. It is shown that reductions in the number of service invocations can be substantial when the called service is part of a client initiated search algorithm. Performing the search on the service side greatly reduces the number of required service invocations. As a concrete example, this paper describes work being performed by CERDEC C2D (Communications Electronics Research, Development, and Engineering Center – Command and Control Directorate) at Fort Monmouth, NJ to expose a military mission plan as a web service through the use of simple SQL-like (Structured Query Language) statements optimized for mission data query.

1. LOOKING TO THE FUTURE

The Army (and in fact the Department of Defense as a whole) is moving towards a Net-Centric world where any soldier can get any information at any time. The Army is in the process of developing infrastructures, such as the Defense Information Systems Agency's (DISA) Net-Centric Enterprise Services (NCES). NCES is comprised of a set of Core Enterprise Services (CES) that act as the foundation for the Service Oriented Architecture (SOA). Among various jobs, these CES provide important service discovery and data mediation functional components. Essentially, the CESs answer the questions, "Where is service X?" and "How can I talk to service X?" The ultimate goal is that systems will be able to access any information from any place at any time.

1.1 The Problem with Net-Centricity

At a core level, having unlimited access to any data you could ever possibly want is incredibly valuable. Commanders in the field will be able to solve problems and get answers to questions that they never could before. However, access to all this data comes at a cost. The most obvious issue is bandwidth availability. Doorways will be opening into previously "stove-piped" systems at an amazingly fast rate. The available bandwidth, however; may not be able to keep up the pace and SOAs (such as NCES) will need to make accommodations for operating in "bandwidth constrained" environments (Pane and Joe, 2006).

Another issue involved with creating services and hosting them on a SOA is *service granularity*. Service interface designers must continually wrestle with the level at which their service provides its functionality. Too low and you end up with clients having to continually make calls to get all the data they need, resulting in unnecessary overhead. Interfaces designed at too high a level may force clients into getting much more data than the client needs. Receiving more data than is required not only aggravates the bandwidth usage, but also requires the client to spend extra time parsing the received message and throwing away information that isn't required.

One approach to addressing interface granularity issues is to meet with all potential clients during the design stages and customize the interface methods to exactly suit their needs. This causes several additional problems. First, in a SOA world, you may not know all your potential clients and many of your future clients may not yet exist. Further, even if you could identify all your clients, their needs will most certainly not remain static throughout your service's lifecycle; your interface may have to be adapted as their needs change. While adapting to suit your clients is a good practice, too many changes to an interface will result in broken applications.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **01 NOV 2006**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**

4. TITLE AND SUBTITLE **Using Query Languages And Mobile Code To Reduce Service Invocation Costs**	5a. CONTRACT NUMBER
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **AMSRD-CER-C2-BC Fort Monmouth, New Jersey 07703**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADM002075., The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**UU**	**8**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

Our approach to addressing both the bandwidth and service granularity problems is to define service interfaces in terms of a query language. The flexibility of a query language permits clients to specifically request exactly what is needed from the service and, as a result, reduce the required bandwidth by optimizing the service response and by shifting searching algorithms from the client to the service.

2. QUERY-BASED SERVICE RESEARCH

Prior to detailing our approach to building a query-based service (QBS), we'll provide further detail justifying why there is a need for such a solution.

2.1 Remote versus Local Function Calls

You can think of a call to a service method as essentially a remote function call. It is a request for information or data processing handled outside of your application. There are many reasons for calling a remote service as opposed to calling a local function. Some examples include:

- *Data locality*. Services may be co-located with data or the data may be too large to transport in its entirety. Terrain databases provide a good example of this case.
- *Intellectual property*. Service providers may own the rights to specific algorithms and wish to retain control.
- *Processing power*. Services may run on larger machines. Clients running on less capable hardware can off load computation.
- *Deployment control*. Centralizing service code on servers may simplify the deployment and configuration management associated with the service software.

A fundamental problem with SOAs is the cost of service invocation: calling a service will cost more than invoking a similar function locally. This will be discussed more fully in the following sections. For all their benefits, services are in no way free and as our research has shown, service invocation costs can be orders of magnitude greater than comparable local function calls. As a result, care must be taken in designing the interface to a service in order to minimize the service invocation cost. Further, developers producing service client code need to be cognizant of the invocation costs and design their clients appropriately.

2.2 Service Invocation Patterns

In order to study the cost of service invocations we examined several different *invocation patterns*. It was our belief that typical service invocations are characterized by searching algorithms. Such patterns are frequently found in military command and control (C2) systems. For example, finding units that need refueling requires searching all units for those meeting the "needs fuel" requirements. Likewise, finding a helicopter landing zone requires searching possible areas for one that meets landing zone requirements. The ways in which to architect the communication between a client and service using this referencing pattern are characterized by the following three cases illustrated in Figure 1.

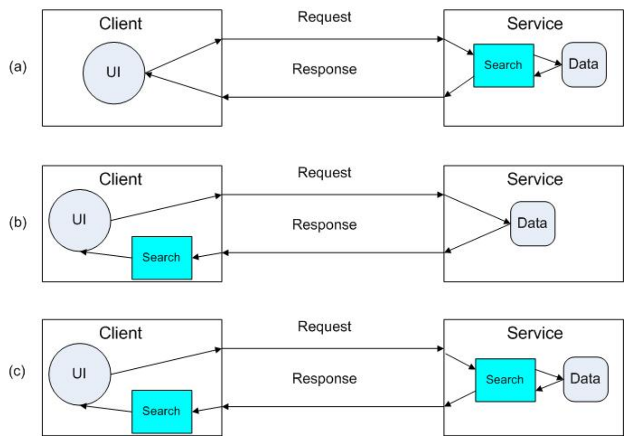

Figure 1. Alternative Invocation Architectures

Case A: *Server Side*. A client makes a request to the service for information. The request is processed entirely on the server side. The server processes the request by searching its local repository for the data requested by the client and then returns the response. An example might be "find all units requiring fuel" and the response would be the set of units.

Case B: *Client Side*. The client requests some (large) subset of all of the information residing on the server. The client then searches through the returned data. An example might be "get all units" and the client then searches for the units that require fuel.

Case C: *Combo*. A blending of cases A and B. The client makes a request for some subsection of the server's data. The server must search through its information and pass some portion back to the client. The client then conducts its own searching and further narrows down the data set to the optimal information.

A case can be made for the appropriateness of each approach. There are certainly times where it makes sense to implement the Combo pattern. Situations like grid computing and shared computational systems often take this approach, dividing computation between the client and the server. However, in most settings, the Server Side is the optimal path. In this case, the service narrows down the data to exactly match what the client is looking for, thus when the client receives the data, there is no additional searching or processing involved. Further, since the search is performed at the server, the client need not perform multiple service invocations, as might be the case in the Client Side approach. Using the server side, there is less wasted data transfer and fewer service invocations.

Our experience indicates that most clients utilize the Combo pattern. The reason is simple: developers find a service that is "close to what they want" and call it. The service returns a data set that has more information than the implementer needs. After receiving the service response, the client has to search through the data for the relevant information. If the service provider is amenable to changing his service, subsequent interface revisions become more and more specialized such that the service more optimally supports the data the client requires. In return, the client will do less processing on the resultant data set. Ultimately, if the interface returns exactly what the client needs, the Combo pattern evolves into the Service Side pattern. This becomes the optimal state because there are no wasted resources: the client gets exactly what it wants in one call.

Clearly this violates one of the premises of SOA in that the service interface is tightly coupled with the needs of a specific client. One of the benefits of SOA is that services are reusable and for reusability to be powerful a service should be usable by more than one client. The solution lies in *providing a service interface that the client can customize at service invocation time.*

2.3 Researching Web Service Overhead

A key premise on which our QBS hypothesis rests is that the overhead of service invocation is substantial. At its most basic level what we're looking for is the amount of overhead (both in processing time and bandwidth usage) that a web service function call adds to the invocation of the given function. In order to explore this, we need to better understand how much overhead is associated with a local functional call and compare that to a similar call over a web service interface. The delta then becomes the net service invocation overhead.

Our initial research looked at the overhead costs for local function calls versus the same function implemented as a service. We chose to implement a squaring function as the function being invoked. Due to the experiment, the actual time to compute the function is immaterial to the results. In order to understand the overhead involved, we can look at the costs in relative terms. Given that the overhead for a local call (O_L) can be defined as the processing time for the call (P) added with the time required to make the call (T_L), we can say that:

$$O_L = P + T_L$$

Additionally, we define the overhead for a service call (O_S) to be the sum of the processing time (P), the time required to make the service call (T_S), and the time required to make the same local call (T_L):

$$O_S = P + T_S + T_L$$

We'd like to determine the overhead associated with implementing a service. Calculating the difference between a service invocation and a local invocation gives:

$$\begin{aligned} O_S - O_T \quad & \Rightarrow (P + T_L) - (P + T_S + T_L) \\ & \Rightarrow P - P + T_L - T_L + T_S \\ & \Rightarrow T_S \end{aligned}$$

Ultimately, we can see that by measuring the difference between the times associated with implementing identical functionality both as a local call and as remote service call, we can determine the processing overhead for the remote call. In essence, T_S is the "wasted" time spent processing a service call devoid of the cost of actually performing the called function. Ideally, we'd like to minimize T_S as much as possible. We'd prefer it if we could have T_S as close as possible to T_L, because that would eliminate the excess overhead associated with service invocation.

2.4 Our Approach to Measuring Invocation Costs

Our research explored the associated overhead costs in terms of both processing time and bandwidth associated with making remote service calls. We implemented the squaring function in both Java and C# and conducted tests both locally and remotely and across implementation languages. The cross implementation experiment was performed because we were interested in the relative efficiencies of Java vs. C#'s implementation of web services. The experiment evaluated the following six cases:

1. Local Java call.
2. Local C# call.
3. Java client to a Java web service.
4. Java client to a C# web service.
5. C# client to a C# web service.
6. C# client to a Java web service.

We used version 1.5 of Java deployed to a Sun Java System Application Server Platform Edition 9. C# was implemented in Visual Studio 2005 with .Net version 2.0 and deployed to Visual Studios internal debug web application server.

We measured both the processing time involved with making local and service calls as well as the bandwidth utilized by the remote service calls. We did this by running varying numbers of requests (from 1 to 1 billion) across varying array sizes (from 8 Bytes to 80 MB, where 1MB = 10^6 Bytes). Each request consisted of a call to a function that calculated the square of each element in an argument array. The array varied in size as described above. Due to processing limitations and time constraints not all pairs of numbers of requests and sizes were completed. For instance, we could only complete a run of 1 billion requests at the smallest size (8B). Because of this limitation, it became apparent that the most useful subset of data was where 100 requests were sent at varying array sizes of 8B, 80B, 800B, 8KB, 80KB, and 800KB. Depicted in the next section are our results for the average of those runs.

The processing time for a local call is defined as the time it takes to complete the following steps:

1. Build data array
2. Pass elements to square function
3. Square elements in array
4. Store results

Processing time for a remote call gets a little more complicated as it includes the above steps, plus everything that is required to implement a service:

1. Build data array (Client)
2. Build service request (C)
3. Send service request (C)
4. Parse request (Service)
5. Pass elements to square function (S)
6. Square elements in array (S)
7. Build service response (S)
8. Send service response (S)
9. Parse response (C)
10. Store results (C)

It should also be noted that because our array sizes increase exponentially ($8 * 10^1$, $8 * 10^2$, and so on), looking at a graph of overhead compared to array size shows deceiving results. For instance, the bandwidth overhead for 800KB is far greater than an 80KB message. While this seems obvious, plotting them on the same chart can lead to erroneous conclusions. As a result, while we'll show both the linear and exponential scales below, one should take caution when drawing conclusions from the linear charts.

2.5 Our Results (Bandwidth)

First, we'll take a look at the bandwidth overhead for service calls. Shown below in Figure 2 is the associated overhead for all 4 types of calls. Note that there is no bandwidth overhead for local calls.

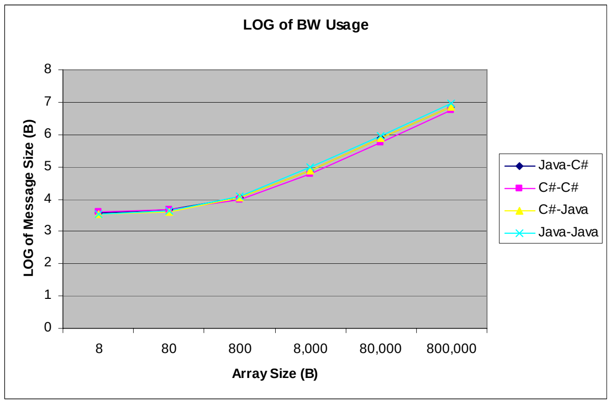

Figure 2. Bandwidth overhead for service calls ($\log_{10}$)

There are several interesting results that one can derive from this chart. First, it's worth noting that all 4 implementations show nearly the exact same results. What this means is that both the Java and C# clients and services use similar service encoding techniques. This demonstrates that both SOAP implementations are nearly the same and are most likely the most efficient. Alternatively, they could both be equally inefficient, but this seems unlikely.

Another interesting note is that there appears to be nearly identical overhead for 8B and 80B, but starting around 800B, the overhead increases steadily. In fact, it never levels off for the data that we are looking at. This essentially means that as you increase your message size beyond some length, you'll continually pay a penalty for the method call.

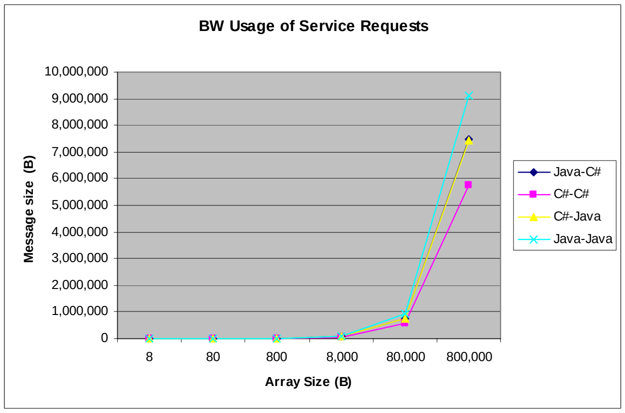

Figure 3. Bandwidth overhead for service calls

Also, the penalty increases as the message size increases. The most likely culprit in this case is probably the fact that the messages are getting packetized at some point around 800B. So, prior to 800B the message is completely contained in one packet. Beyond that point, they are getting packetized and subsequently have additional header and footer information added; therefore, the overall message size increases.

The non-logarithmic scale (Figure 3) is also interesting. Obviously, on the exponential scale, the results are much more dramatic, but there is a visible jump in message sizes. However, it only shows up above 8KB on this chart. What is more noticeable in Figure 3 is that there is a significant overhead in the Java implementation that is not present in the C# implementation at the higher end. In fact, C# shows about a 30% improvement on bandwidth efficiency. The C#-Java and Java-C# implementations fall in between the others. This may be a result of slightly differently constructed SOAP messages on the part of the servers.

The important point here is that if we are aware of the point where packetization takes place and modify our request to fall under that threshold, we could substantially reduce the amount of traffic on the network. The problem is that as a client, you are limited to the interface that the service provides you. As a result, you have no direct method to modify the way that you communicate with a service. In order to process a request, a service requires a certain amount of information. As a client, if you fail to provide some of that information, the service does not have to provide a response. In the traditional sense of services, the client cannot dictate what and how much information it sends in a request. An important feature of our proposal is that if a service is defined with a query-based interface, clients can include as little or as much information as they need to put in the request.

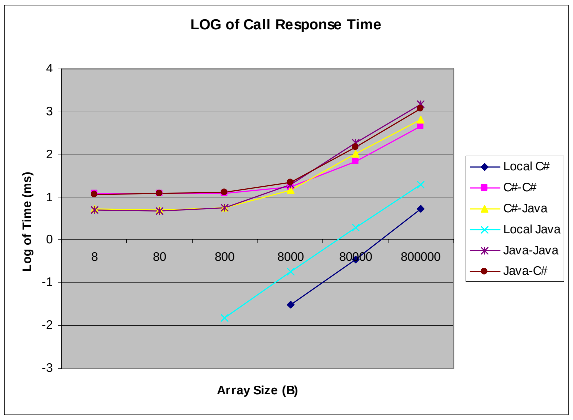

Figure 4.Call Overhead Time ($\log_{10}$)

2.6 Our Results (Processing Time)

The next set of results relates to the processing time required to handle a request. Again, this shows the average processing time over 100 requests for different array sizes. We'll include all 6 sets of tests here (see Figure 4).

This chart shows us the overall turnaround time (form request to response) of 6 different implementations of service calls. Two of them are strictly local and the remaining four are remote.

The first thing to note is that obviously the local calls (Local Java and Local C#) have much faster response times than remote calls. When comparing Java remote to local calls (for an array size of 800KB) our T_S value is 1.48 seconds. This means that you can save 1.48 seconds off of your processing time if you make the function call local instead of remote. This may seem meaningless until you begin to think of this is terms of multiple services and servers processing multiple requests. In a Net-Centric world, services will be operating in a communal state, meaning they'll all be relying on each other, and processes running through multiple services will become commonplace. This means that when a user makes a request, that message could be broken up and passed through multiple services serially. In other words, first data goes to service A, then the output of service A is fed to service B and so on. In this case, any time you can shave off of each service call, benefits the entire chain as a whole.

The next thing worth mentioning is that (as with the bandwidth data) you can see a jump in processing time around 800Bs of data and higher. Again this can be attributed to packetization. If clients were aware of that threshold and could create their requests in such a way as to avoid the threshold, the time savings would be substantial.

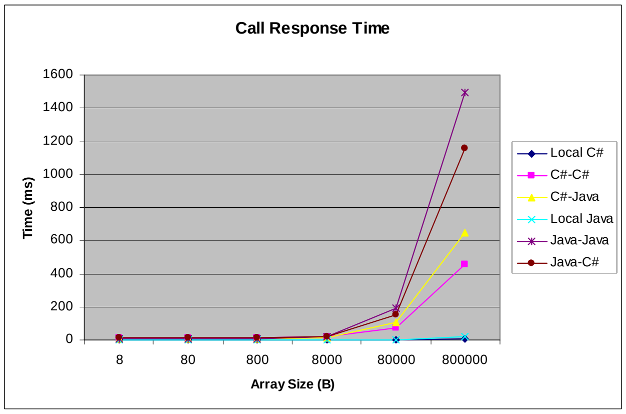

Figure 5. Call Overhead Time

Also, interesting is that it seems that with relatively small data sets, Java appears to be more efficient. The Java-Java and C#-Java implementations stand out as processing the results faster than their service-based counterparts. Beyond the 8KB mark, C#-C# seems to be the clear winner. This can be more clearly seen when viewing the data on a non-logarithmic scale (see Figure 5).

When viewing the results on the exponential scale, the differences become much clearer. At an array size of 800KB, the rank from most to least efficient service process is C#-C#, C#-Java, Java-C#, Java-Java. In fact, the difference is so drastic that a straight C# implementation beats out a straight Java implementation by over a second. For some reason, the combination of the function we chose to implement as well as the software components used, resulted in a situation where, given a choice, C# would be the optimal implementation language.

While this research effort wasn't focusing on implementation questions, it has become apparent that the service implementation language (or SOAP libraries) may have something to do with processing time. As a result, when designing the QBS, this is something we should take into consideration. Regardless, there certainly is interesting information that could be gathered in this area. It would be worthwhile to explore how implementation language, SOAP libraries, XML parsing libraries, and server types affect the processing overhead time of a service call.

2.7 Our Results (Conclusions)

The intent of the above research was to provide a foundation for the proposal of a QBS. In order to understand the intent of the QBS, one must understand why there is a need for it. What we have shown above is that there is without a doubt, an overhead in both processing time and bandwidth usage when comparing local system calls to remote service calls. Also, the overhead can be substantial and there are certain "sweet spots" that should be targeted.

Ultimately, the point of the research was to showcase what happens when bandwidth and processing time is wasted. We'd like to get to a point where we're residing completely in the Server Side pattern (see Figure 1) and this is where our interface is at its most optimal. We are not proposing that services are bad because they waste resources. However, what we are proposing is that when designing service interfaces, they should be flexible and customizable. The next section of this paper outlines the concepts for the QBS approach.

3. QUERY BASED SERVICES

The foundational concept for the QBS approach is to design service interfaces that accept as an argument statements in a language that describe what the service is to return. If services can be designed in this way, then the services can perform any operation that can be represented in the language. The net result is that clients can indicate to services precisely what they expect the service to provide so that the interaction between client and service becomes optimal as defined in section 2.2 above.

There is historical precedence to this approach in current database technology. Virtually every commercial database system may be thought of as having a service architecture based on query statements. In modern databases clients obtain a connection to the database, send a SQL (Structured Query Language) message to the database and then retrieve the results. The client is able to obtain exactly what it needs in (notionally) one request. The result is our *Service Side* pattern.

3.1 Two Case Studies

The initial question that needs to be answered when using QBS is whether the service you are building lends itself to a language based interface. Database applications are appropriate based on experience with SQL but what other types of services are amenable to this approach? We experimented with two different applications and can form some generalizations from them.

The first application is a plan service based on the Combined Arms Planning and Execution System (CAPES) developed by the Command and Control Directorate (C2D). CAPES produces a *plan data model* that is a representation of the expected state of a military mission over the mission lifetime. An ideal candidate for QBS, the application is a plan query service that will provide information about the plan to any requesting client. A client could, for example, request the expected fuel level on 1CAB unit at mission time H+3 hours. The reason a service is ideal to provide mission data is because high-resolution military plans are difficult to represent as pure data. The cause for this difficulty is that time variant functions must be either stored as algorithms (not data) or as sampled data points. If sampled points are stored, then an algorithm must be used to interpolate between samples. The net is that some form of algorithm needs to accompany the data. Using a service as the home for the algorithm is ideal.

Initially the CAPES plan service was designed to be a family of services that answered different typical questions about the plan. Some design effort went into this approach but it quickly became apparent that the number of services was expanding explosively. The design was switched to a query based effort and the results were much more manageable. The approach used to implement the plan service is described in section 3.2.

A second system is an alert management service. This service receives alert specifications from clients, monitors situational awareness information and then generates alerts if specific conditions are met. The alert is specified using a query language that specifies the alert conditions. In point of fact, the alerting service implements two languages: one for alert conditions and the second to specify the actions to be taken when the alert triggers.

Both of these examples illustrate that non-database applications can benefit form using a QBS approach. The CAPES plan query service has a database feel to it, but unlike a strict database, the CAPES system combines algorithms along with the data. The alert application is clearly not a database but instead is a specification language in which clients can indicate what is to be monitored.

In our opinion it is safe to say that any service could be implemented using the QBS approach described here. The reason we feel confident making that statement is because the SOAP messages sent to invoke services are actually a type of language, albeit a very simple language that has only one action (the service invocation), but a language none the less.

3.2 Implementing Languages

Our experience has shown that the definition and implementation of the language is the most daunting task in the adoption of QBS. In the CAPES query system one designer, after seeing the requirements announced that the effort to build the query system would exceed a man year just for the language! The actual level of effort in this case was more like 2 man months.

Applying the QBS concept proceeds along several predictable steps that include:

1. Designing the language.
2. Implementing a parser.
3. Implementing an interpreter.

The following discusses these steps in some detail and describe our experience in their implementation.

Designing the language is the "art" portion of QBS application. The challenge is this: examine a range of interesting problems then design a language that makes representing the solutions to the problems elegant. We found in both of our test cases that the language was represented as a collection of operators and arguments. Arguments fed the operators which performed an operation on the argument and then produced a result that could be fed to additional operators. Statements in the languages can easily be represented as trees with the operators being the nodes of the trees. Consider the expression A = B x C + D. This statement can be represented in a tree form as shown in Figure 6.

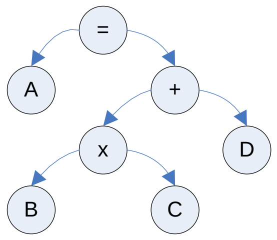

Figure 6. Tree Representation of A=BxC+D

Representing statements in the language in this form is straight forward in that each node can be an instance of an object representing *nodes*. In our languages, each node had an abstract method called `Evaluate`. Calling `Evaluate` causes the node to evaluate its sub-trees and return the results of the node. For example, in the figure evaluating the "+" node would cause it to evaluate the left sub-tree to get the product of B and C and then evaluate the right subtree to get the value of D. The "+" node would then return the value of B x C + D to the next higher node.

Using a tree of operator nodes to represent the language has a number of useful benefits. First of all, it is easy to extend the language to add additional features. Language extension means adding additional nodes. This is an interesting feature in the context of QBS because it is possible to extend the service interface without breaking the calls made by existing clients. The idea is that the new language features are added to the existing features rather than replacing them.

A second benefit of the tree representation is that it can be easily represented in XML. The tree from Figure 6 can be converted into the following XML

```
<Equals>
  <Value>A</Value>
  <Plus>
    <Times>
      <Value>B</Value>
      <Value>C</Value>
    </Times>
    <Value>D</Value>
  </Plus>
</Equals>
```

This means that it is possible to use a QBS approach without having to bother with actually parsing the language representation. Using trees to represent the language and using XML to represent the syntax avoids all the mess of writing a mini-interpreter. It is up to the client to produce the language statements in the proper format. If a client chooses to implement a more conventional (human readable) infix notation for the language, then that is the client's decision.

The third benefit from representing the language as a tree is in the simplicity of the evaluation. As described each node has an `Evaluate` function. The `Evaluate` function in our implementations is located in a Node base class so all of the derived nodes provide an implementation of this function. Calling the top level node in the tree effectively performs a recursive tree walk that is controlled by the individual nodes. For example, in our implementation of the CAPES query language we have an "*and*" node that mimics the && function in C++. Our implementation of the && evaluates its left sub-tree first, looks at the result and if the result is false, then the right sub-tree is not evaluated and a false is returned. Having each node control the evaluation process for itself simplifies the interpreter design yet provides a sophisticated implementation.

The fourth benefit of the tree structure is that algorithms can be written that optimize the evaluation of the tree. For example, an algorithm can be written that searches for common sub-trees and only evaluates these once even though they appear in the overall tree several times. In an application such as our alert service where statements in the language are repeatedly evaluated (the alert conditions), tree optimization can be a very useful tool for reducing the computational load.

3.3 Mobile Code

If a QBS can accept a single statement in a language, then why not allow it to accept multiple statements? The concept would be to allow the client to produce small snippets of program that could be passed as an argument to the service and then executed on the server. Such an idea goes by a number of different names: agents and mobile code being two of them (*mobile code* is probably the less pejorative of the two given the recent bad press agents have been receiving).

One concern brought up by the concept of mobile code being used as a service interface is how to best protect the service from malicious code. It is conceivable that an enemy (or poorly designed client) gaining access to a service could send a piece of mobile code that would ask the server to calculate Pi to the last decimal place. Services that support mobile code would have to be designed to prevent such problems. Perhaps the service would support a *mobile code docking site* that provides an interface to the mobile code that offers up the requisite functionality without compromising the server itself.

Microsoft's managed code concept running under .NET goes a long way to define how services might be able to produce reliable docking sites for mobile code. The .NET application domain hosting technology currently used by the MS IIS Web server to safely host multiple web applications on the same server could be applied to mobile code (Löwy, 2003).

Whatever the technology, mobile code would provide the ultimate in server flexibility and permit small clients to offload complex computational tasks to servers.

4. CONCLUSIONS AND FUTURE RESEARCH

Our initial investigation into using languages to define service interfaces has been very positive. Our research into the cost of service invocations indicates that anything that can be done to reduce the number of calls to services and to reduce the amount of returned data was an effective way of lowering overhead in a SOA.

We found that implementing a QBS was straight forward and that much of the implementation technology could be reused in different applications. Further we found that QBS could be easily extended without breaking clients that were already in existence.

Our investigation suggests a number of interesting topics for additional research. Among these are:

1. How to use mobile code to build service interfaces.
2. How to develop a toolkit that simplifies the development of query based services.
3. How to better quantify the types of services that would benefit from QBS.
4. How to define rules to help in the development of the languages used in QBS.

REFERENCES

Pane, J.F. and Joe, L., 2006: Making Better Use of Bandwidth: Data Compression and Network Management Technologies, http://www.rand.org/pubs/technical_reports/2006/RAND_TR216.pdf, xi-xii, 1, 4.

Command and Control Directorate (C2D), 2005: CAPES Users Manual.

Löwy, J, 2003: .NET Components, *O'Reilly & Associates*, Inc., 234-245.