Automatic Subscription and Notification in Event-Driven Web Services

Vijay Dheap and Paul Ward

November 3, 2003

We first describe the motivation for event-driven web services, and then outline our approach to the problem of automatic subscription and notification.

1.1 Motivation and Observation

First consider the case of Mary who plans to pick up her mother from the local airport. After arriving at the airport Mary discovers that the flight was delayed by 45 minutes, long enough that she might have done something else in the meantime, but now that she is at the airport, there is little she can do but wait.

What could Mary have done? First, she might have 'phoned the airport before leaving work early. This suffers from two deficiencies.

1. The onus is placed on Mary, and if she forgets, she still suffers the wait.
2. In general the 'plane is not late. Mary should not have to 'phone to find out that "things are normal."

The second solution would be for the airline company to set up a notification service. Mary can register to be notified in the event that the flight is delayed or canceled. This solution, while solving the second of the above-listed problems, still puts the onus on Mary to register for the notification. If she forgets, she is back to the original problem. Worse, if she remembers, but then her mother changes her plans and decides to take a taxi, Mary must remember to deregister the notification. The third possibility is to have the travel agent register the notification on Mary's behalf at the time that the flight is booked for Mary's mother. Presumably the travel agent would also perform the deregistration when the mother changed her plans.

This initial scenario is relatively simple, involving a single notification request, that can probably be registered by a human in a fairly straightforward manner. Now consider the case of Simon in Connecticut for three days on business. On his last day he gets caught up in a seminar and when it is over he realizes that he has to rush in order to make his flight back home. On his way to New York it starts raining and as he approaches the airport flights are getting canceled due to the deluge. Upon reaching the airport and giving up his rental car Simon realizes that his plane is canceled. This requires him to make his way into New York City to find a hotel for the night, since the hotels near the airport are all booked. He would have to either rent another car or use the cab. He will also have to make arrangements with the airline for an alternate flight.

In this case the notifications are considerably more complex. If Simon had been notified as the information became available, he might never have returned the rental car, and possibly could have extended his hotel booking in Connecticut. If he has already left Connecticut when he is notified of the flight cancellation, he requires instead a hotel booking in the nearest available hotel. Thus, not only must Simon be notified of the problem, but so must the car rental company and either the hotel he was staying in must be notified, or a new hotel room must be procured, and that decision will be context dependent. Finally, a new flight must be acquired through the airline reservation system.

There are several observations that can be made about this scenario. First, it should be clear that the notification scheme has rapidly become too complex for a typical human user to take advantage of the system. Not only would Simon have to register interest the status of his flight, but so would the car rental company, the hotel, the airline reservation system. This would still leave open the problem of what to do in the event that Simon has already checked out of his hotel and needs a different hotel. Second, while creating an application to deal with this specific scenario is not difficult, such an application would have no further use beyond this immediate problem. Who then would spend the time to create such an application? Third, although the scenarios have been travel-based, there is no reason why the problem is limited to that domain. Any environment in which events trigger changes in actions that require human intervention can cause such problems to occur. In the normal course of events a certain process is followed. In the case of a divergence, various entities should be notified pro-actively. Any workflow processes will fall in this category, though it is clearly broader than this.

We may then summarize the main justifications for constructing an event-driven web service architecture as follows:

1. An individual can manually register for notification in trivial scenarios, though this will still require some level of technical expertise, which may be lacking, as well as putting the onus for notification on the individual. In any non-trivial scenario the complexity is likely beyond all but the most sophisticated of users. However, most scenarios are too specific to warrant the writing of an application. A generic solution is called for.
2. An event-driven model applies to more than just travel scenarios. It can be used in any environment in which events trigger changes in previously prescribed actions. Normally a certain process is followed. In the case of a divergence, various entities should be notified pro-actively. Thus, the approach can be used in health-care management, taxation systems, business process management, conference organization, *etc.*.
3. Events cross organizational boundaries. Any automated system to deal with events must likewise be able to cross such boundaries. This implies that the architecture must be loosely coupled.

Given the justification for a loosely-coupled event-driven approach, we make the following observations:

1. A mechanism is needed to automatically subscribe to other relevant web services. Services are relevant if they are connected to the current agenda of this web service. The web services subscribed to are expected to detect any significant events which may influence the current or future agenda of this web service.
2. If such events occur, the web services subscribed to should notify this web service. Such notifications should be processed automatically by this web service. This may require policy specification (*e.g.* what does the traveler wish to do in the event of flight cancellation?), though a suitable default policy may be possible in most instances.
3. Web services are loosely coupled. Transactions, by contrast, are tightly coupled, and are therefore a poor match for web services (WS-Transactions notwithstanding). What is required is an approach

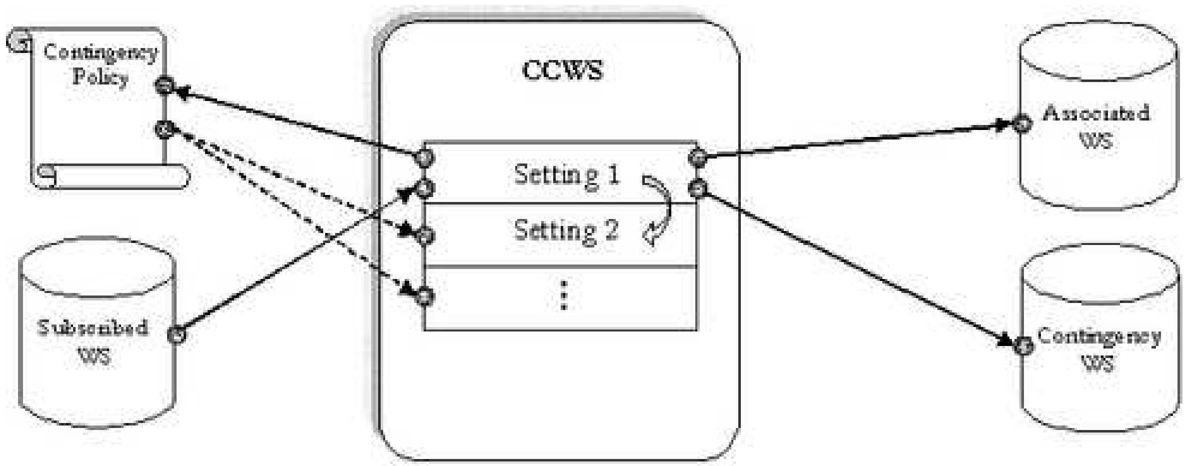

Figure 1: Proposed Architecture

more in keeping with a fault-tolerant view, in which no single web service prevents progress. In this manner composed services can provide the required level of QoS.

1.1.1 Constraints on Solutions

In a loosely-coupled system controlled and maintained by multiple parties with diverse (and possibly conflicting) interests, it becomes necessary to consider the deployment strategy of a particular solution. In this regard we identify the following key constraints:

1. Only universally accepted facilities (*e.g.* HTTP, TCP, XML, SOAP, *etc.*) may be employed in the solution.
2. The solution should only require one point of acceptance. In other words, a party interested in using this solution should be able to do so without having to convince other parties to adopt it or provide auxiliary support. That said, like Lego™ building blocks, the more web services that use our approach, the more interesting things that can be built.

1.2 Solution Approach

Figure 1 shows our proposed high-level view of the event-driven web services architecture.The key features of our approach are as follows.

The Context Capture Web Service (CCWS) is the core component of the system. It is a web service whose function is to keep track of the current and future context (the Settings) of any client web services. It is required to provide its service for an arbitrary, though pre-defined at invocation, period of time for any given client service. The service provided by the CCWS is to notify subscribed client services when events occur in their current context or changes will be required in their future context. This would be augmented by an automated response service to anticipated changes.

The Settings in the CCWS are discrete contextual instances that are being tracked by the CCWS. Each instance is linked with a set of Subscribed Web Services (SWS), Associated Web Services (AWS), and Contingency Web Services (CWS). An instance also has a Contingency Policy (CP). The client web service of the CCWS is always within exactly one Setting. The Settings are ordered, such that the CCWS will always be able to determine, based on the current environment, and any events that have occurred, what Setting the client service should be in.

The SWS is a set of web services that a Setting element subscribes to so that it can receive notifications when events occur in its context or that may potentially influence subsequent Settings. The AWS is a set of web services that is directly relevant to the current context environment of the Setting element. It is important to understand the difference between these two elements of the proposed architecture. The SWS are those services that are subscribed to because of the agenda of the client web service. The AWS are those services that are invoked (possibly *via* subscription, though not necessarily) by CCWS because of the current environment of the client web service. Thus, if we consider the traveler web service, the airport flight information will be part of SWS. By contrast, the airline WS will be part of the AWS.

The CWS is a set of web services that may be invoked upon an event in the Setting's context (received through a notification). This set is not necessarily exhaustive but a cache of the relevant web services that may be required to respond to events as outlined by the Contingency Policy.

The CP is a policy that determines how the client web service of the CCWS wants to respond to anticipated event notifications. If an event occurs that a policy has not been specified for then the default contingency service will be invoked to deal with the situation interactively with the CCWS. The policy may influence or require modifications to subsequent Settings of the CCWS. It is our hope that the major aspect of "application development" within our environment would be Contingency Policy specification and agenda establishment in the CCWS.

1.3 Research Issues

There are several open research problems that arise in our proposed architecture. Some of the key issues are the following:

1. Automatic subscription and notification
2. Web service transactions
3. Continuous service
4. Automatic composition
5. Determining context
6. Defining the contingency policy
7. Caching relevant contingency web services
8. Providing for and specifying human user interaction

Our preliminary research identified that there is currently no general model for automatic subscription and notification. The recently published WS-Events has some useful capabilities in this regard, though it is still quite limited, and largely similar to existing (though proprietary) systems. In particular, it is tightly coupled, and not generic. Further, communication is transient, not persistent. We require a generic subscription and notification mechanism, that can be applied and processed automatically by arbitrary web services.

The contingency policy and and associated services are key to our proposed approach to the problem of automated response to event notification. At present, notifications can only result in the user being informed of the event (other than in tightly coupled systems, which require an individual application for each scenario). We believe that a number of events can be accurately anticipated and the user only needs to specify the desired response to those events in advance of their occurrence. Even in this case, a large number of default policies will likely capture most user intent. For unanticipated events a default contingency policy will be invoked, which (likely) will bring human intelligence (not necessarily the user) into the loop.

The remaining issues can be resolved separately, with suitable defaults assumed for our current purposes. This does not negate their existence, or the fact that these need to be resolved.

1.4 Proposed Action

We propose to implement a subset of the architecture as a proof-of-concept. Our work would focus on the automatic subscription and notification aspects. When this implementation is complete, we will create at least two solutions within diverse application domains. One of the domains will be that of the traveler. The traveler will have a web service exposed calendar, which will be the client to the CCWS. The expected outcome of our approach is that the traveler will be able specify just the contingency policy, and the remaining system will deal with the subscriptions and notifications and operations as events occur.

The second application domain has yet to be chosen, but is required to be quite different from that of the travel industry. This is intended to demonstrate that the architecture is general, and not specific to the problem at hand.