High-Level Design of Event-Driven Web Services Architecture

Vijay Dheap and Paul Ward

1. Purpose

The purpose of this document is to provide the high-level design of our proposed architecture for enabling dynamic event-driven interactions in web services. It identifies the main components and the relationships among them. Preliminary interfaces have been outlined. Information flows during operation are also documented. We start, however, by describing a typical scenario for which we expect this architecture to be used. Our high-level design will then be connected to this scenario through the document.

2. Scenario

Adam schedules a trip to New York from Boston. Having booked his trip using his corporate web-service travel planner, he would like his itinerary kept in his corporate calendar. In current web services, such an operation will typically have to be performed manually, representing a repetition of input of (a subset of) the same data that Adam must provide when initially making his travel plans. Such redundancy yields only error and reluctance to use calendar systems.

Further, if circumstances should force Adam to make changes, he would like those changes incorporated into his calendar also. For example, if a flight he is scheduled to take is canceled, it would be useful if, at the very least, he was notified of the problem. Better yet, rather than receive a notification, he might simply be happy if a new flight was arranged. Likewise, if he changes his plans, and makes such changes in his calendar, it would be useful if the implications of these changes were enacted upon.

The key point in this scenario is that Adam would like the system to keep track of the implications and changes necessary. Adam merely wished to enter his plan once, and have the system track any environment changes. Likewise, any changes that Adam makes to his schedule should propagate to any dependent systems.

This document describes a high-level design of our proposed architecture for such reactive event-driven web services.

3. Design

The high-level view of our architecture is illustrated in Figure 1. At the heart of the system is the Context-Capture Web Service (CCWS) which is, in essence, a structured calendar that is able both to subscribe automatically to relevant web services and act as a service itself, being both query-able and able to generate notifications of change. It maintains an agenda for the various users that subscribe to its service.

To know how to react, the system maintains a set of contingency policies that are executed when notifications are received. These are perhaps best thought of as an exception-handling mechanism. As with exception handling, contingency policies form a hierarchy,

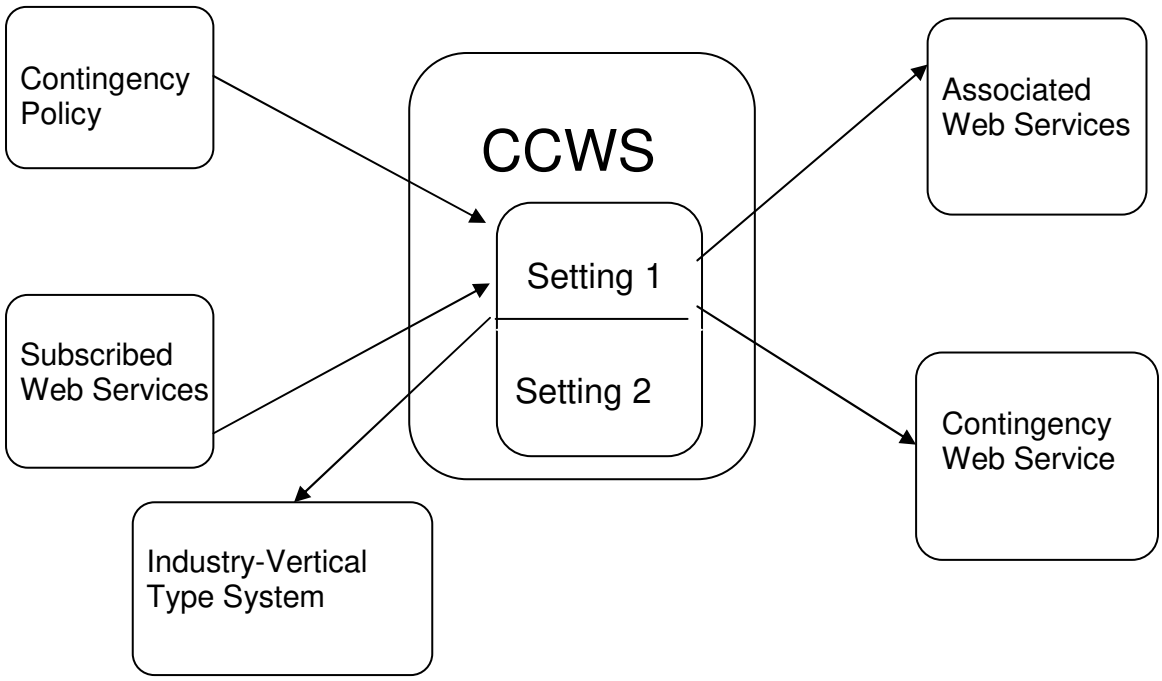

Figure 1: High-Level View of Architecture

with low-level ones associated with specific events, while more general ones exist to catch problems that are either not anticipated in detail or do not need a specific solution.

While contingency policies tell the system how to react, actions taken are enabled by knowledge of relevant services. These are the associated and contingency web services.

The system knows what to react to by subscribing to various relevant services. This is the set of subscribed web services, which are expected to generation notifications when changes relevant to the agenda occur.

Finally, to ensure that the system is not tied to our specific travel scenario, all operations are specified over an Industry-Vertical Type System (IVTS). These are perhaps best thought of as class types systems in object-oriented languages.

Thus, in the case of our scenario, Adam will use a travel-plan web service to first book his trip. This service is invoked *via* the CCWS system. In this way the agenda can be imported directly into CCWS. In particular, the associated web services (that is, those web services used to create the agenda) are known. Since it is a travel situation, the industry-vertical type system used to interpret the data would be the travel one. This will contain such information as the fact that each flight has an associated airline, what the departure and arrival times are, *etc*. In this way, the subscribed web services may be determined. For example, we expect the travel type system to indicate the appropriate web service to subscribe to in order to receive notification of flight changes and cancellations. Finally, the contingency policy would specify the actions to be taken in the event of change. For example, in the case of flight cancellations the default policy might simply be to notify the user (in this case, Adam), and offer to cancel all dependent items in the schedule.

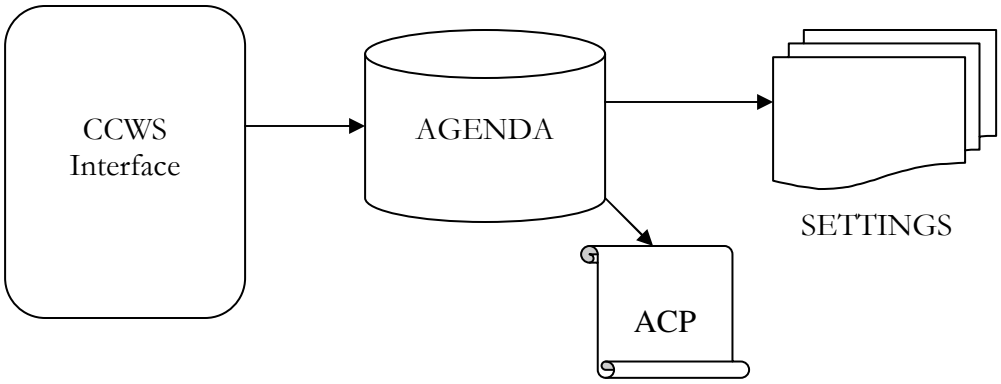

Figure 2: CCWS Detail

We now describe in more detail the various subcomponents of the system. We start with the CCWS component, as it is the heart of the system.

3.1. CCWS

The Context-Capture Web Service (CCWS) is the central component in our architecture. CCWS is the engine that coordinates all the activities to support event-driven interactions. It monitors context on behalf of clients to enable them to adapt automatically to events in their environment. CCWS is at heart a structured calendar, knowing the user's intent, and thus easily able to determine deviation from that intent.

The high-level design of CCWS is illustrated in Figure 2. It comprises the interface, the agenda, which is a series of structured settings, and the agenda contingency policy.

3.1.1. Interface

The interface provided by CCWS to the client is the following:

1. *Create:* Establish a CCWS service instance. The instance created receives a handle from the invoking entity so that it can send updates.
2. *Insert:* Add a new setting to the agenda.
3. *Delete:* Remove a setting from the agenda.
4. *Update:* Change a setting in the agenda.
5. *Subscribe:* Register for notification of changes. Subscription may be to the agenda as a whole, or to an individual setting.
6. *Unsubscribe:* Remove a prior subscription.

3.1.2. Agenda

The agenda is a collection of "settings" established by the client to be managed by the CCWS. A setting is a task to complete together with (knowledge of) the services and policies required to accomplish that task. The agenda is a structured calendar in that the settings have a formal type (defined by IVTS, rather than being simple text) and are partially ordered according to dependency. In our example of Adam taking a trip to New York, his car rental, hotel booking, and return flight would all be dependent on his initial flight. Should he cancel his initial flight, CCWS should notice the dependency and cancel the dependent items.

In addition to the basic interface described in Section 3.1.1, settings may be accessed by time of day and by content.

3.1.2.1. Settings

A setting is a tuple with the following content:

1. *Start Time*: The time at which a setting is scheduled to begin (if meaningful).
2. *End Time*: The time at which a setting is scheduled to be end (if meaningful).
3. *Type*: The type of the setting, specified by the IVTS type system.
4. *SWS*: Web services subscribed to for notification in the event of change.
5. *AWS*: Web services used to create this setting, including input to said services.
6. *STS*: Settings subscribed to within this agenda.
7. *SCP*: Contingency policy specific to this setting (if any).
8. *Priority*: self-evident.

While several of these items are self-evident, a few require additional comment. The start time and end time may only be meaningful for tasks such as flights or meetings, which have a clearly scheduled beginning and end. Other tasks may have nominal start and end times. For example, hotel room bookings have specific check-in and check-out times. However, it is rare that people actually match those times. Finally, some tasks simply do not have well-defined start or end times, but are rather items that simply have to be done at some point. Adam, while in New York, might plan to visit his mother. However, that may well be performed based on a spur-of-the-moment 'phone call, rather than on a clearly planned schedule.

The setting has a formal type which we will discuss in Section 3.2. In the case of Adam's trip, there will be several settings, including one for the flight to New York, one for a car rental, one for a hotel booking, and one for the return flight. These settings will be defined as subtypes of the more general travel type.

The subscribed web services are those services that this setting has subscribed to so that it may determine if there is any relevant change that affects the user. Thus, for Adam's trip, the setting for the flight to New York will have an SWS that contains (at least) a subscription to the relevant airline so that notification can be given in the event of flight delays, cancellations, *etc*. The SWS may also contain subscriptions to the source and destination airports, and possibly other sites. The full SWS may be dependent on the contingency policy of the client, as what is relevant to one individual may not be relevant to another.

The associated web services are maintained to allow for change and removal of the setting in question.

The settings subscribed to within the agenda are required to tie together collections of individual settings into a single operation. Thus, when Adam travels to New York, he has a return flight, a car rental, and a hotel booking. In the event that the setting for the flight to New York is deleted or updated, the settings for the other items may need to know of the change.

Finally, each setting may have a contingency policy specific to that setting. Details of the format and operation of these policies will be deferred to Section 3.1.3. It is sufficient to note that this policy is a specification of the response required to notifications. For any

given setting type there may be a default policy specified by the IVTS. However, such a default policy is not used at this stage. Rather, if there is no policy specified in this setting for a given notification, then that notification is handled according to the agenda contingency policy.

Continuing the example of Adam's trip, the contingency policy for his car rental may include some details about what to do if the car he desires is not available. Such a policy is not meaningful at the level of the agenda, but only at the level of the car setting.

3.1.3. Agenda Contingency Policy

We first define the format of contingency policies. We then describe the order of their processing. Finally, we note that the contingency policy database must have an interface, and so we describe what that must be.

Contingency policies, be they setting-specific, type-default, or agenda, are described by an event-condition-action (ECA) rulebase. The event is the notification that is received. All notifications must be typed, *per* the IVTS. Matching is based on the type, in the same manner as exceptions may be processed in languages with exception handling. The condition is the state of the environment that must be true for the rule to be accepted. For example, rules to process flight-cancellation notifications may differ according to whether the user is still at home, or if s/he is already away. The action is will be in the form of a sequence of web services to be invoked. We now describe how notifications are processed.

If there is no setting-specific policy to deal with a particular notification, that notification is processed according to the agenda policy. The most-likely initial processing choice of the agenda policy is to see if there is a type-specific default and, if so, invoke it. This is a sufficiently likely case that it forms the default policy of the agenda. However, while it is a reasonable choice, the user may have a preferred alternative. For example, the default in the event of a flight cancellation may be to schedule the next available flight with the same carrier. The user may prefer instead to switch carriers. Since the choice is that of the user of the system, notification handling is passed to the agenda policy first, rather than to the type-specific default. We note that it is likely that corporate users of a system such as ours may which to override the IVTS type-specific defaults. We expect to incorporate such a mechanism within the notification handling subsystem shortly. Finally, if there are neither setting-specific nor type-specific policies for the notification, then the agenda default policy is invoked. This policy is, unless altered, a simple user-notification policy.

The interface to the policy rulebase consists of four operations:

1. *Insert*: Add a rule to the rulebase.
2. *Delete*: Remove a rule from the rulebase.
3. *Update*: Change a rule in the rulebase.
4. *Find*: Locate a rule in the rulebase.

3.1.4. Basic Operation

We now describe the actions of the CCWS when performing the various tasks described in Section 3.1.1. We focus on the actions of insertion and deletion of settings. Update is merely deletion followed by insertion, suitably optimized to remove redundant actions.

Creation, while essential, does not directly pertain to event-driven services. Subscription by others simply requires that notifications be issued when actions corresponding to items of interest occur.

Insertion into the agenda must perform the following tasks:

1. *Create SWS*: The set of web services that are to be subscribed to must be identified and subscribed to.
2. *Create AWS*: The web services associated with the creation of the setting must be identified.
3. *Create STS*: Any settings that this setting is dependent on must be identified and subscribed to.

We deal first with the creation of AWS, as this is a straightforward matter. These services are identified by invoking them through CCWS. That is, CCWS will be expected to have knowledge of all items used to create a setting merely by catching the requests as they flow through CCWS. The WS-Routing protocol can be used to provide this capability.

The SWS set is determined by the IVTS of the setting, together with the various contingency polices. We expect IVTS to explicitly identify the formal web services that must be subscribed to, and these will be accepted by default. Thus, the travel IVTS will have a specification of the flight type. The flight type in turn specifies the airline service that must be subscribed to. The details of the particular service to be subscribed to would then be determined by the combination of this type information and the specific airline that is be used.

We expect the IVTS to identify services that must be subscribed to, as well as some that may be of interest. For example, airport information may be merely interesting, but not essential. Determination of which services to subscribe to from this set would be policy-based. Additional services may be required, also based on policy. For example, corporate policy may require subscription to corporate travel advisories when traveling.[1] While the creation of SWS might be complex in a traditional text-based agenda, it is relatively straightforward in our structured agenda. Further, there is no ambiguity or chance based on semantic interpretation of the calendar contents.

Finally, we must determine dependencies within the agenda. At present this is an open problem, since it cannot be solved by application of a general type system or by policy. Rather, it will require knowledge of the user's connected operations. Currently we require the insertion operation to specify (the transitive reduction of) all such dependencies.

Deletion of a setting from the agenda is a matter of reversing any external operations performed by insertion. Thus, any services subscribed to (including internal settings) must be unsubscribed. For setting subscriptions, a notification is set to perform the unsubscribe operation. In this way, the setting subscribed to may choose to act on the unsubscribe operations as a notification to be processed.

---

[1] While we have identified the need for such a policy-base, we have initially presumed that this would be a part of the contingency policies. This is probably not a good choice. We are evaluating alternatives.

3.2. Industry-Vertical Type System

The IVTS is crucial to the correct operation of our proposed system. It is left as an industry vertical, since it requires domain knowledge that is not generally available outside the industry segment. The authors of this design document currently diverge on how best to implement the IVTS. There are two approaches under consideration: type system (class hierarchy) and ontology.

In the type system approach, an industry segment formally specifies the type hierarchy for their segment. Thus, the case of Adam's travel plan, a flight will be a type within this hierarchy. A flight "is-a" method of transport from A to B. The more general type would specify the requirement for A and B. Those items would in turn have types. Thus, if we have a flight type, then A and B will be airports. There may also be intermediate stops in the trip.

In addition to specifying this hierarchy, the type system must specify the various services that can be associated with any given type. Thus, a flight would have an airline carrier. That carrier would have a notification service for its flights. Since the hierarchy is well-defined, these subscriptions are not guessed at, but rather computable. This is a highly-desirable feature in this approach.

The ontology approach attempts to use existing research on ontologies (*e.g*. DAML+OIL, *etc*.) to determine relevant subscription services and contingency services. An ontology is available, together with knowledge of the specific domain that the setting refers to. As such, the vocabulary is interpreted according to the ontology. This approach has the possibility of being more flexible than the type system. However, it has the drawback that it is less clear how it would function, or whether it would correctly identify the relevant set of services to subscribe to. We suspect it may be of greater value when determining STS.

4. Scenario Revisited

We now complete our design by reviewing the operations that will be performed for Adam's trip. To review, Adam schedules a trip to New York from Boston. His agenda, that he wants his CCWS service to keep track of on his behalf, is as follows:

1. 9:30 AM – *Flight 301 American Airlines* Departure: *BOS* Arrival: *LGA* Class: *Business* Duration: *60 min*
2. 10:30 AM – *Commando Limo Service* Limo: *Lincoln* Pickup: *LGA* Drop-off: *Plaza Hotel* Duration: *20 min*
3. 10:50 AM – Hotel Check-in Room: Presidential Suite (Plaza)
4. 12:00 Noon – Goldman Sachs Partner Meeting Location: Plaza Conference Room Duration: 180 min

Adam may use any client device to access the CCWS service. We presume he imports the agenda in from another application, in this case the travel application Adam used to plan his trip. The IVTS for each setting is determined at the point when the setting is being inserted into the agenda. Since the application from which the data is imported was associated with the travel industry, the type of each setting is determined by the travel IVTS.

Some of the key words identified by the IV are shown in italics. The CCWS would use these key words to subscribe to relevant services when building the SWS, which are given below.

1. American Airlines Flight Info service, BOS and LGA airport info service
2. Commando Limo Update Service, Plaza Hotel info service
3. Plaza Hotel Room info service
4. Goldman Sachs corporate info service (This may require authentication which is user specific. The user may also specify private UDDI registries to search.), Plaza Hotel Room info service

The CCWS would also subscribe to additional services that match the IVTS of the Agenda (*e.g.* travel advisory, weather service, *etc*.).

The AWS would maintain the set of web services that were used to make the booking by the tool. Below is the AWS for each setting.

1. American Airlines Flight booking service
2. Commando Limo reservation service
3. Plaza Room booking

The CP either needs to be specified by Adam or it must be imported from the tool used to make the travel arrangements. For example, the planning tool may have requested first and second choices for the hotel reservation or company meeting. Specification of the rules remains a complex issue. Specifically, it is not feasible to specify a rule for every contingency. The example rules presented below are a rudimentary first step to addressing this.

1. Event (Flight Status = Canceled); Condition (Location = Boston); Action (Search for Flights Departure: BOS Arrival: LGA and Display results to user).
2. Event (Flight Status = Delayed); Condition (); Action (Notify user; Seek alternate flights; Generate delay notification)

As we can see, the notification of the flight delay generated in the second rule will inform all subsequent dependent settings of the delay, and they can act as required by their contingency policies. Thus, the rule is kept relatively simple, while allowing the effect of the delay to propagate to subsequent agenda items.

The ACP specifies global default rules (which can be customized by the user) for events that are not handled by a setting CP. An example of such a rule might be "On receipt of a Travel Advisory Event, e-mail the user the Advisory message."

The CCWS searches using the keywords in CP rules for services that will be required to respond to notifications:

1. Flight-schedule service, hotel-booking service, room-booking service, *etc*.
2. Services in the AWS will need to be employed if a rule in the CP is executed. For example, giving up a seat in the American Airlines Flight.

To better understand the information flows in the CCWS system let us analyze what happens when changes to the agenda trigger adaptation.

Case I

1. Travel advisory event received by CCWS from the Travel Advisory subscription service.
2. CCWS searches SWS to find the first (based on time or priority) setting that is subscribed to that service.
3. The SWS record for the subscription service reveals that it has been subscribed to because of its relevance to the Agenda.
4. CCWS searches the ACP for an adaptation response to the event.
5. Using one of the global defaults it sends an e-mail to the user of the message extracted from the travel advisory event.

Case II

1. American Airlines Info event stating that the flight has been cancelled due to airplane maintenance is generated by the American Airlines info service.
2. CCWS searches SWS to find the first (based on time or priority) setting that is subscribed to that service.
3. The SWS record shows that setting "A" is subscribed to that service.
4. CCWS searches the CP of "A" for a rule specifying the adaptation response.
5. The adaptation response specifies that other flights be searched with the same departure airport and destination airport.
6. CCWS searches the CWS for a service that provides flight search and booking functionality.
7. If a service reference in CWS is found then browser based interface (JSPs) can be generated for the user to use that service to delete the current setting and replace it with a new one.
8. If a service reference is not found then a more comprehensive search using the setting's IV. Subsequently the user would be provided with a browser based interface (generated automatically) to replace the current setting.
9. Let us say that the user selected to book United Airways Flight 202 with a departure time at 10 am.
10. All settings that were subscribed to the initial task are notified of the changes.
11. Similarly the adaptation-response cascades using the CP of each setting to make appropriate changes.
    - An event will be delivered to the next setting (Limo) stating that the flight will be 30 minutes late. The setting will follow its CP which may require notifying the AWS of that setting so that the limo can arrive at the new time.

V. Development Environment

This project will be implemented using J2EE web services technologies. The tools that will be employed include WebSphere Application Server, a version of IBM's UDDI Server implementation, and the WebSphere Application Development studio.