

SCI Group CCA Event Specification

Nathan Dykman, Kostadin Damevski, Ayla Khan, Steven Parker
3/28/2007

1 Introduction

This document contains a proposal to add event processing capabilities to the CCA specification. The goal here to provide a simple mechanism for event distribution that meets the needs of the HPC community, while adhering as much as possible to other commonly used frameworks and taking full advantage of their experience and ideas.

2 Issues in Event Management

The most difficult issue to address in event service design is that it must work in an HPC environment. In this environment, it is vital that the developers have ultimate control over the amount of resources that the event system uses in the implementation. Additionally, the event system must only impose overhead if it is used. Finally, it must integrate well with other CCA services.

2.1 Basic Event Management

The model we propose is that of a “publish-subscribe” model. In this model, events are organized into a namespace of topics. Each topic has its own set of properties that determine how events on that topic are published, distributed, and received.

At the heart of the event system is the event management service. This service is responsible for organizing the namespace of topics, setting and enforcing event policies, and maintaining the resources needed for event distribution. This single service is accessed through two interfaces to differentiate the methods that ought to be called by an event publisher from those called by a subscriber.

Events need to be handled either asynchronously or synchronously. A dedicated execution resource (most likely a thread) needs to be available to the event management service in the case of asynchronous event management. In the synchronous case, the component must either yield to allow events to be processed or to allow events to be handled as they are raised.¹

2.2 The Publish/Subscribe Model

The model that this specification adopts comes from the Message Oriented Middleware domain. There are various MOM systems in use today.² At the heart of most MOM systems is a system that provides a hierarchy of topics on which events can be sent (publishing to a topic) and or received (subscribing from a topic).

Topic hierarchy organization is analogous to a file system structure. This hierarchy, which allows topics so be grouped together, allows one to subscribe to topic groups as well as individual topics.

¹ In this model, raising an event will cause all the event subscribers to be called and processed. Of course, this means that the event raising may never return.

² For example, the JMS specification is responsible for providing MOM-like services to J2EE applications.

Here is are two sample topics:

```
functions.signals.currentStepDone
functions.signals.currentStepDone.time
```

From this, we can tell that there is a topic group called `functions`. In that group, we have a sub-group of topics called `signals`, and in that group, there is another sub-group of topics called `currentStepDone`. Finally, there is a topic called `time`.

To listen for all events on every topic in the `functions.signals` group, we can subscribe to one or more topics using a wildcard syntax. Note we use a “*” as the wildcard token to match either of the sample topics given above.

```
functions.signals.*
```

To subscribe to a single topic type, for example

`functions.signals.currentStepDone`, use a “%” as the wildcard token.

```
functions.signals.%
```

Even more flexibly, we subscribe to any group which has a subgroup called `signals` as follows³:

```
*.signals.*
```

It is vital to note that topics are just channels in which events are sent. The topic does not bind what type of event can or can’t be sent on it.⁴ In this sense, the topics are merely named channels for events to be distributed, nothing more.

Also, it is important to note that a subscription can match one or more topic channels. The kind of matching described above is fairly flexible.

2.3 Event Structure

Our events will have the following information:

Message header: This contains the delivery information for each event: basic metadata useful for every event. What this contains at the very least is the destination of the event. More realistically, this would contain information about the sender of the event, an event ID, and so on. In the previous draft, we had a separate section for message properties. In this version, the header contains both standard and non-standard metadata about a message. A message header should not be empty.

³ This kind of flexibility does add to the complexity of the runtime. If needed, wildcards can be most-specific only; meaning that after a wildcard, no other wildcard or topic can be listed.

⁴ Of course, one extension is to allow for typed topics. However, given that topics are filtered, one can create a filter to give the same results as a typed topic.

In this specification, a header is TypeMap of keys and values. All reserved header properties have keys that start with an underscore. End users should not create or overwrite these properties.

Message body: This is body of the event. In this specification, we allow for a CCA TypeMap body type.

2.4 Topic Extensions

There are three potential extensions to the topic model to be considered in this specification. The first is to allow a topic to be local to a single component. The second is to have a set of filters associated with a topic. Lastly, is to allow a topic to be persistent. We discuss each extension in turn.

- Local scale: This means that the events are only seen on a per component basis, only on a per-framework basis.
- Filtered components: This allows for filters to be run on the events when they are sent or received. The filtering runs on the event-service resource, not on the resource that is sending or receiving the events.
- Persistent topics: This extension allows topics to be queued when sent. This allows subscribers to receive all the topics sent to a topic, even if the subscriber didn't exist when the event was sent.
- Time-to-live: Allow events to expire.

2.5 Event Scale

In the previous draft, there was a discussion on event scale. After further discussion, we have decided on a much simpler model. The event service is only responsible for distributing events to listeners in a framework. It is left up to each component to decide how to further distribute events based on the framework and/or component implementation.

2.6 Reserved Topics

In CCA, it is very likely that there will be a predefined set of topics, each with a specific purpose. As such, the top level topic **cca** is reserved for standard CCA events.

2.7 Topic Management

Crucial to any MOM system is who can create and delete topics. One workable model is to only allow components with the proper permissions to programmatically create topics.

A more realistic model tracks the resources associated with a CCA task or computation, and gives each of these tasks their own set of resources⁵ (including queues, etc.).

Of course, the system should come with tools that manage topics, queues and the event system in general. These tools will allow management of the topic space, logging and monitoring of the event system and management of topic queues.

In our current draft, the process of topic management is handled solely by the event service itself. Since current services in CCA have singleton semantics, this is an effective place to put the topic management methods.

3 Draft SIDL

This draft is currently based on a prototype event service implemented in the SCIRun2 code base. It is greatly simplified from the original draft.

```
interface EventServiceException extends CCAException {
}

interface PublisherEventService extends cca.Port {
    cca.Topic getTopic(in string topicName) throws EventServiceException;

    bool existsTopic(in string topicName);
}

interface SubscriberEventService extends cca.Port {
    cca.Subscription getSubscription(in string subscriptionName)
        throws EventServiceException;

    void processEvents() throws EventServiceException;
}

interface Event extends sidl.io.Serializable {
    cca.TypeMap getHeader();
    cca.TypeMap getBody();
}

interface EventListener {
    void processEvent(in string topicName, copy in Event theEvent);
}

interface Topic {
    string getTopicName();

    void sendEvent(in string eventName, in cca.TypeMap eventBody)
        throws EventServiceException;
}

interface Subscription {
    void registerEventListener(in string listenerKey, in EventListener
theListener) throws EventServiceException;
```

⁵ Actually, the topic of resource allocation, control and security is a very interesting one in HPC that goes beyond this.

```
void unregisterEventListener(in string listenerKey);  
  
string getSubscriptionName();  
}
```

4 Issues in Implementation

At the heart of the implementation is the process events call on the event service. This is when all event processing occurs. In a system that doesn't allow for a dedicated resource to be scheduled to process events (a dedicated thread, component in a given framework, etc.) then the computational resources must yield explicitly by calling process events on the event service.

An efficient implementation of an event service is not easy. However, even simple implementations can be efficient if the following guidelines are followed:

- Do as little computation in event receivers as possible. The event receiver should note the appropriate state, store it, and return. Complex logic or other processing should be done elsewhere.
- In systems with multiple threads, the task of event processing can be parallelized by using a work queue model. For each event that needs to be processed, it can be handed to a thread in the pool. When the processing is finished, the thread returns to the pool.
- If a dedicated set of threads can be used for event processing, then each thread can maintain its own queue of events. When each event is sent, it is delegated to each thread in a round-robin or other scheduling mechanism. A more advanced mechanism could dynamically size the thread pool based on demand.

5 Additional Areas of Inquiry

This model is very oriented around the notion of topic-based event management. Another popular model is one that is present in many UI frameworks. In this section, we discuss how .Net explicitly supports these kinds of events by using delegates to process events, and providing explicit event declaration.

A delegate can be best thought as a type-safe function pointer. In this model, instead of having to provide a class that implements the `EventListener` interface, you just have to wire an type-compatible method to the delegate. In .Net, one can provide a static method (or course), or an instance method, which is a pointer to the method call on a specific instance.

The main thing to note is that delegates can be chained together, so that a single event can be distributed to multiple receivers.

An event is merely an asynchronous call to an event. Any class can declare an event of a given delegate type. To raise the event, you merely call it as if it were a function. To respond to an event, you merely add a method that matches that delegate type to the

event. The model is fairly simple and, in this author's opinion, much simpler and easier to use than the corresponding anonymous inner classes technique used in Java.

However, it is unclear to me that this can be done in all the languages that CCA needs to support, and clearly, this requires considerable runtime support.