

Using The O₂S Component Framework

MIT CSAIL, O₂S Group

Contents

1	Introduction	1
1.1	Motivation	1
1.2	System Concepts	2
1.3	System Model	3
1.3.1	Entity	3
1.3.2	Registry	4
2	Getting Started: Code Walk-Through	4
2.1	Introduction	4
2.2	Code	5
3	Running Code	9
4	Supplements	9

1 Introduction

This document describes usage of the O₂S Component Framework. The O₂S Component Framework provides an abstraction layer and framework for developing distributed, network objects. The System Model section describes the basic concepts of this framework; the Code Walk-Through is a tutorial that illustrates a simple application using the framework. Familiarity with Python is assumed.

1.1 Motivation

In order to achieve fault-tolerance and adaptability in distributed, dynamic computing environments, the O₂S Component Framework provides a programming model that separates each application into two distinct layers: (1) a distributed network of interconnected modules performing application-generic computations, and (2) application-specific logic that assembles that network and monitors its subsequent operation. The first is a system whose execution produces the actual behavior required of the application and is optimized for performance, while the second is a *meta*-application whose function is to configure, monitor, and potentially reconfigure the first (Figure 1).

Specifically, the application-specific logic runs in a simple, typically single-threaded environment, using a simple API to construct a “circuit diagram” of module instances interconnected by typed data streams. During operation, the application-specific logic monitors and reacts to a stream of high-level events, including notifications of failures and new resource availability.

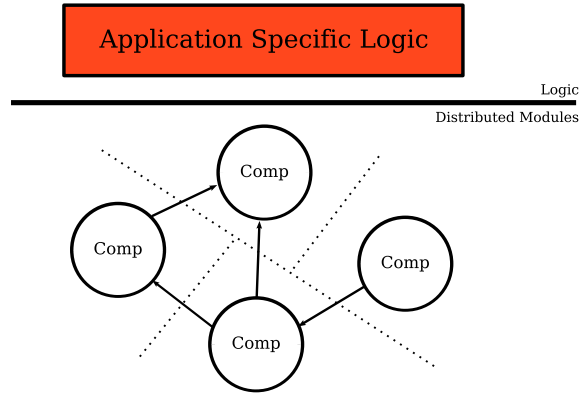


Figure 1: System Architecture View: separating application logic from distributed computational mechanisms (labeled “comp”).

1.2 System Concepts

There are three major concepts in the O₂S Component Framework: *Network Objects*, *Events*, and *Connectors*.

Network Objects These refer to software objects (which contain methods and state) that can be passed to remote hosts. Remote hosts can then invoke method calls on these network objects. Typically a network object will represent a resource or device in the world (e.g., a pair of speakers) and implement a coherent set of functions to allow remote hosts interact with the resource.

To develop a network object in the O₂S Component Framework, you first develop an ordinary object; to enable remote access to your object, you simply wrap your object in the Network Portable Object Packaging (NPOP) framework. You can transform any ordinary object into a network object by using NPOPs. NPOPs follow a client/server model: when an NPOP network object is passed to a remote host, only a handle (automatically generated) is passed to the remote host. When the remote host invokes a method call on the handle, the call is forwarded to the server hosting the network object. Generation of handles and data marshalling happens automatically within the NPOP framework.

While there exists many Remote Procedure Call and distributed object packages, NPOPs provides a simple, lightweight framework for rapid pervasive application development under a variety of platforms and languages.

Events While network objects and the NPOP technology provide a framework for building and accessing distributed objects, all procedure calls on these objects are generally synchronous in nature. Events provide a general and light-weight mechanism for sending asynchronous notifications between network objects. In O₂S, the event handling framework is also wrapped with NPOPs, thus enabling Events to be passed between remote hosts.

Stream Connectors O₂S *Connectors* differ from both network and Events. Whereas the latter two are highly structured for controlling distributed objects, Connectors provide a faster, stream-oriented connection between network objects. Connectors are more appropriate for streaming high bandwidth data between two network objects (e.g., sending MP3 audio from a file server to a handheld speaker device).

Connector objects by themselves represent a dedicated network socket on a host. Connectors are also wrapped using NPOPs, thereby enabling remote applications access to appropriately connect and disconnect the network socket to sockets on other hosts.

A summary of the concepts is listed in Table 1.

Concept	Data Flow	Usage
Network Objects	Synchronous	Building and controlling remote objects
Events	Asynchronous	Message Passing
Connectors	Stream	Sending high-bandwidth data streams

Table 1: Summary of concepts.

1.3 System Model

The O₂S Component Framework also consists of the *Planner*, *Entity* and *Registry* (see Figure 2). The remainder of this document discusses the Entity and Registry; the Planner is beyond the scope of this document.

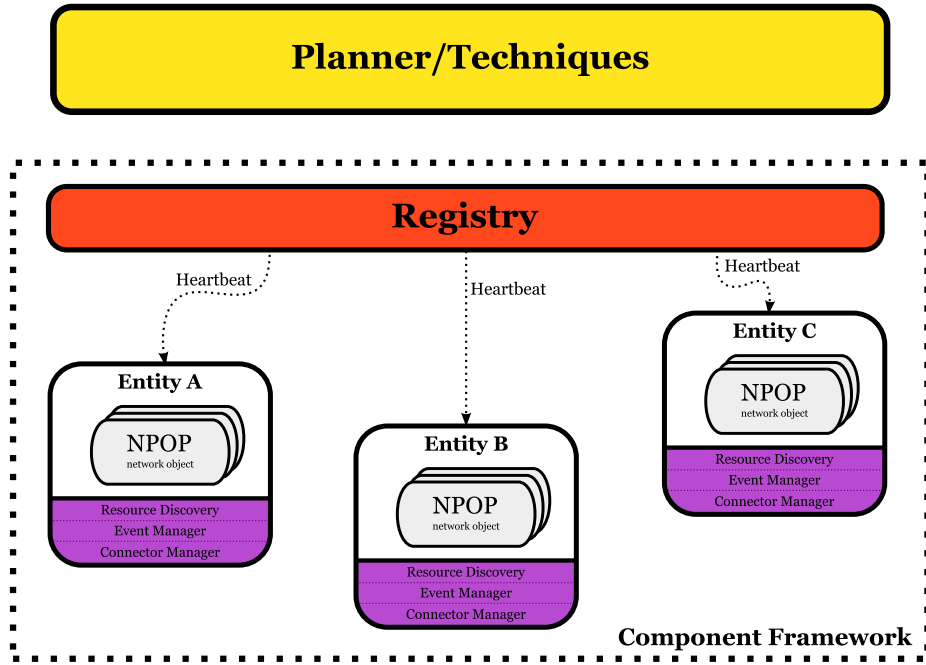


Figure 2: Component View: Overall System Diagram. NPOP network objects are hosted on Entities, which provide these objects with various run-time services such as resource discovery, Events management, and Connectors management. The Registry performs liveness monitoring on all Entities via a periodic heartbeat.

1.3.1 Entity

The Entity represents a logical host in the O₂S network and provides a runtime environment for NPOP network objects. The Entity's basic functions are to:

1. Register with the Registry; the O₂S Registry monitors the Entity's liveness (health) and notifies other hosts when an Entity's liveness changes.

2. Manage one or more NPOP network objects. The Entity provides a runtime environment for NPOPs, providing them with various services such as Event Listeners (to receive Events), stream connector management, and resource discovery.

Once you develop your NPOP network objects, you'll use the Entity to host, manage, and run these network objects. It should be noted that Entities themselves are network objects and wrapped with the NPOP framework, thereby allowing Entities to be passed to remote hosts.

1.3.2 Registry

The Registry provides liveness monitoring for all Entities in O₂S via a periodic heart-beat. Network objects can *subscribe* to be notified (via Events) about changes in another Entity's liveness. This is particularly useful for network objects that depend on services provided by other network objects on different hosts. For instance, if network object *A* depends on network object *B* (running on a different Entity as that of *A*), *A* could subscribe to be notified when the Entity hosting *B* fails (and/or recovers).

2 Getting Started: Code Walk-Through

In this walk-through, we will develop a few network objects using NPOPs and illustrate the use of Events and stream Connectors.

2.1 Introduction

When the members of MIT CSAIL moved into their new home at The Stata Center, many researchers were delighted with the motion sensors in their offices: when the sensors detect a person entering the office, the lights automatically turn on. Let's say that in addition to turning on the lights, we'd also like the room to start playing some music to set the mood. Let's also assume that: we have access to the Motion Detector device; that we keep our repository of songs safe on a separate file server in the server room, which we'll call the "Jukebox"; and finally, we want the music to play from the speakers on our desktop, which we'll call the AudioSink. Figure 3 illustrates the setup and the procedures.

The application first contacts and registers with the MotionDetector, so that the motion detector will notify the application when the a person enters the room. Here is what will happen when a person enters the room:

1. The detector sends an asynchronous Event to the Application.
2. The application finds the Jukebox and AudioSink and requests streaming connectors from these network objects
3. The application connects the two Connectors to form an audio stream, and finally,
4. The application invokes the Jukebox to start streaming audio through its Connector.

Important: recall that Connectors represent dedicated network sockets and are wrapped with NPOPs; hence, Connector objects can be passed to the Application. By connecting the Jukebox Connector to the AudioSink Connector, the Application is actually instructing the network socket on the Jukebox to connect *directly* to the network socket on the AudioSink.

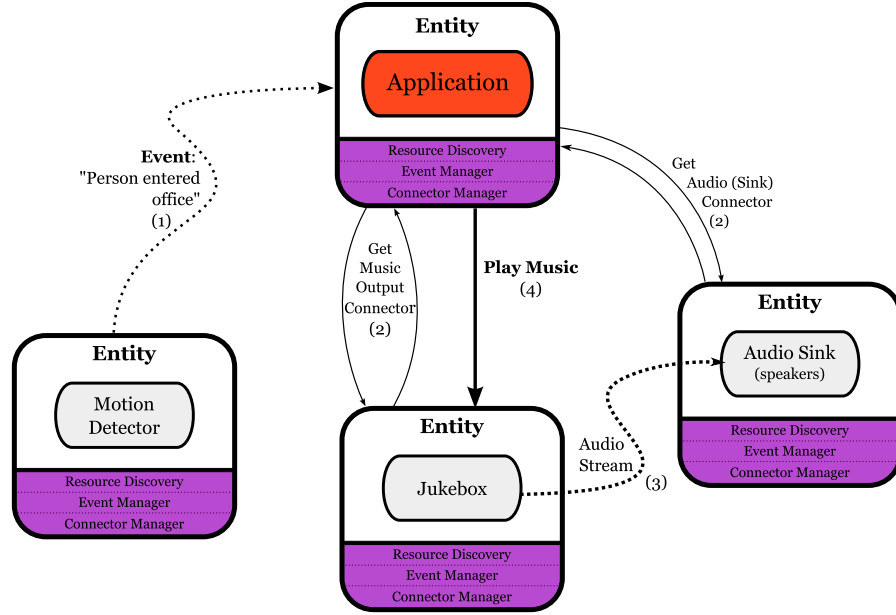


Figure 3: Application Schematic (Registry not shown). When the application receives an Event from the Motion Detector (triggered by a person entering the room), the application obtains Connector objects from the Jukebox and the Audio Sink. It connects the two Connectors together to form an audio stream and subsequently instructs the Jukebox to start playing music.

2.2 Code

Application Let's first look at the application. The Application illustrates the tasks of resource discovery (searching for other Entities in the system), making procedure calls on these network objects, and connecting together Connectors.

```

1  class Application(Entity):
2      def __init__(self):
3          # initialize our base class
4          Entity.__init__(self,
5                          register = False,
6                          entity_type = "Application")
7          md_profiles = self.find_entities_of_type("MotionDetector")
8          md_profile = md_profiles[0]
9          self.md_npop = md_profile['resource']

```

We subclass Entity because the Application will ultimately need to receive Event messages (recall that the Entity provides asynchronous Event handling functionality). Additionally, Entities provide methods to discover other Entities; in this case, line 7 searches for the `MotionDetector` Entity. `find_entities_of_type()` will return a (Python) list of profiles that describe Entities matching the name `MotionDetector`. (The profiles are represented as Python dictionaries; the NPOP network object for the `MotionDetector` Entity is in the `resource` key).

```

10         # now get an event listener from our entity
11         # and pass in a handler
12         self.event_listener = self.get_event_listener(callback_handle =
13                                                         self.handle_event)
14         # and tell the motion detector that we want to know
15         # when someone walks through the door
16         self.md_npop.registerToBeNotified(self.event_listener)

```

Because we wish the `MotionDetector` to notify the Application when a person walks through the door, we need to make an *Event Listener*. The Event Listener is a network object that we will pass to `MotionDetector`; the Event Listener represents a link for the Motion Detector to relay events back to the Application. Line 11 gets an Event Listener, provided by the Entity class. The argument `callback_handle` specifies the method that will be called when a new event arrives. In this case, the method `Application.handle_event()` will be called when the Motion Detector sends the Application an event:

```

17     def handle_event(self, incoming_event):
18         print "Got Event: %s" % incoming_event
19         if incoming_event.get_message_type() == "Motion Notification" \
20             and incoming_event.get_data() == "ENTER":

```

In the event handler, the first argument must always be the received event. We first check the Event to make sure it's from the Motion Detector (see below, where the Motion Detector generates this Event). If this event is from the Motion Detector, we should play some music:

```

21         entity = self.get_entity()
22
23         # now query for the Jukebox
24         jb_profiles = entity.find_entities_of_type("Jukebox")
25         # take the first one, and get its npop
26         jb_npop = jb_profiles[0]['resource']
27
28         # now query for the Speakers (AudioSink)
29         as_profiles = entity.find_entities_of_type("AudioSink")
30         # take the first one, and get its npop
31         as_npop = as_profiles[0]['resource']

```

Similar to before, we search for the Jukebox and the AudioSink (speaker) Entities (lines 22-30). Because the Entity provides resource discovery services, we first obtain a handle on our Entity (line 21). Once the application obtains handles to the requisite network objects, the application can then query for their respective stream Connectors:

```

32         # get connectors
33         musicSrcConnector = jb_npop.getMusicSource()
34         audioOutputConnector = as_npop.getAudioConnector()
35         # connect these connectors
36         Connect(musicSrcConnector, audioOutputConnector,
37                 self.event_listener)
38         # tell jukebox to start playing
39         jb_npop.playSong('Dave Brubeck - Take Five')

```

Finally, the application uses the `Connect()` method to connect two stream connector objects. The third parameter of `Connect()` (line 37) is optional; if an Event Listener is specified, the connectors will send events to notify the application when the connectors disconnect or other error conditions. Finally, the application tells the Jukebox to start playing music.

Motion Detector Now let's look at the Motion Detector object. This object illustrates the task of generating Events and sending them to the Application.

```
1  class MotionDetector(RService):
2      def __init__(self, Entity, Location):
3          # do specific initialization first
4          self.Location = Location
5
6          # list of event listeners from other
7          # entities who wish to know when
8          # someone walks through the door
9          self.interestedParties = []
10
11         # then initialize RService base class
12         RService.__init__(self, Entity)
```

First, we subclass `RService`; this automatically wraps `MotionDetector` into an NPOP.¹ When initializing the NPOP, be sure to initialize the base `RService` class as well (line 12). In line 9, we initialize a list to keep track of the Event Listener's from various applications interested in learning when someone walks into the room.

```
13     def o2s_registerToBeNotified(self, event_listener):
14         # let's make sure only one registration!
15         print "Got registration: %s" % event_listener
16         if event_listener in self.interestedParties:
17             raise Exception, "Hey, you've already registered!"
18         self.interestedParties.append(event_listener)
```

The application code (line 15) calls method `registerToBeNotified` on the Motion Detector object; the method itself is implemented here. Notice that 'external' methods—that is, methods which can be called from remote places (such as the Application)—are pre-pended with `o2s_` in their name. However, when invoking these methods, you may drop the `o2s_` tag in the method name.

The application passes in its Event Listener, and so the Motion Detector adds it to the list of Event Listeners. Note that you can and should `raise` (throw) exceptions in error conditions, as these exceptions will be thrown back automatically to the Application.

```
19     def o2s_getLocation(self):
20         return self.Location
```

Here is another example of an external method, `getLocation()`, which returns the a string describing the location of the motion detector.

```
21     def personWalkedIn(self):
22         # make an event
23         ev = Event(message_type = "Motion Notification",
24                     thrower = self,
25                     recipient = None,
26                     message_string = "Someone just entered",
27                     data = "ENTER",
28                     parameters = {'time': time.asctime(time.localtime()),
29                                   'location': self.Location})
30         for each_listener in self.interestedParties:
31             each_listener.throw_event(ev)
```

¹The NPOP base class is named `RService` for historical reasons.

We assume that the Motion Detector driver will call method `personWalkedIn` whenever the hardware detects an arriving person. We'll first generate an Event object; there are no set types for the Event fields: the fields are all application-specific. However, it is important that the Application know beforehand the data types to expect in these fields. Finally, in lines 30-31, we retrieve each Event Listener registered, and send the event back to each respective Application.

Jukebox The Jukebox is relatively simple; it exports one output connector.

```
1  class Jukebox(RService):
2      def __init__(self, Entity):
3          RService.__init__(self, Entity)
4          self.audioConnector = Entity.make_connector(name = "audio_output",
5                                                    direction = "out")
6
7      def o2s_getMusicSource(self):
8          return self.audioConnector
```

The Jukebox exports one method that returns its output connector. Line 4-5 asks the Entity to generate an output stream connector, and Line 8 returns the Connector object to the application (or any other requester).

```
9      def o2s_playSong(self, name):
10         # fetch music name, and send it over the connector
11         print "Playing: %s" % name
12         try:
13             self.audioOutputCon.send(name)
14         except Exception, e:
15             print "Could not send data! Perhaps audio output disconnected."
```

After the application connects the Jukebox to the Audio Sink, the application tells the Jukebox to `playSong`. The above method is called, and in theory, the Jukebox would fetch the song from disk and stream it to the Audio Sink. For sake of brevity in this example, we'll instead just send the song name to the Audio Output. Note that if an exception is thrown in the `send` method, this indicates that the connector is disconnected.

AudioSink The AudioSink is very similar to the Jukebox. AudioSink illustrates the use of input stream connectors:

```
1  class AudioSink(RService):
2      def __init__(self, Entity):
3          RService.__init__(self, Entity)
4          self.audioConnector = Entity.make_connector(name = "audio_input",
5                                                    direction = "in",
6                                                    handler = self.audioInput)
7
8      def o2s_getAudioConnector(self):
9          return self.audioConnector
```

Like the Jukebox, the AudioSink provides a method to return an input Connector. However, because the AudioSink object needs an input Connector, we need to specify an additional parameter `handler` for `make_connector`. This argument specifies the "callback method" to call when the Connector receives input (in this case, `AudioSink.audioInput`).

```
10     def audioInput(self, audio, connector_object, meta):
11         # play audio here, but for now, we'll just print it
12         print 'Got audio: %s' % audio
```

In the stream input handler, the first argument (`audio`) should always be the received data. In theory we should pipe the audio out to the sound card driver, but because the Jukebox merely sends the song name over the connector, the `AudioSink` will simply print it to screen.

The second argument (`connector_object`) is a handle to the connector which received the audio data. This can be useful if the `AudioSink` object (ultimately) provides multiple connectors which share a single input handler. The third argument provides additional meta-data for the stream, which might contain information regarding the audio stream (e.g., sampling frequency, sample width, number of channels, and so forth).

3 Running Code

So far, we've developed several NPOP network objects—but how do we put them into action and run them?

Using The Entity In the above examples, NPOP network objects often used `get_entity()` in order to request Event Listeners, stream Connectors, or discover other resources in the system. To instantiate and execute NPOP network objects, we'll need to create some Entities to host them: for this example, we'll create one Entity per NPOP object—although in general, a single Entity can manage many NPOP objects running on that host.

Here's an example of an Entity for the Motion Detector:

```
1 class MotionDetectorEntity(Entity, MotionDetector):
2     def __init__(self, location):
3         Entity.__init__(self, entity_type = "MotionDetector")
4         MotionDetector.__init__(self, Entity = self,
5                                 Location = location)
```

We use multiple inheritance in this example by inheriting from `Entity` (which makes `MotionDetectorEntity` a true Entity) and `MotionDetector`—the NPOP network object we just developed. Note that this Entity is also an NPOP object and will register with the Registry so that it can be found by the Application. It is important to initialize both the `Entity` (first) and then `MotionDetector` afterwards. For the `Entity`'s constructor, there are various keyword arguments that can be supplied (which specifies the Entity's profile). In practice, only `entity_type` is required to name your Entity.

The Entities for the other NPOP objects follow almost identically.

4 Supplements

The above code can be found in the O2S subversion repository: `/docs/resources/sample/`.