Jukebox: Services

Table of contents

1. Background

Early experiments with the threads in `OS/2` showed that the concept of a thread is weak where it gets to providing reliability and consistency. Threads offered by host operating systems were useful, but handicapped: they didn't offer any kind of fault tolerance. All the problems that happen just cause the thread to die, without any kind of notification to possible consumers of the information that the threads might produce, thus causing difficult to debug deadlocks.

`Service` was introduced to address this specific problem, as well as to provide uniform lifecycle and platform independent synchronization primitives between producers and consumers in a parallel system.

It turned out to be so useful, all the projects that have been created since are heavily using them.

2. Messenger

This is the simplest service possible, however, it was created the last.

The concept is simple: you create a messenger instance and send it on its way. When the task is completed, the messenger notifies you using the Asynchronous Completion Token. No problem ever gets unreported.

Use Case: Write an audit record to the disk and report back.

3. Passive Service

The passive service lifecycle is as follows:

1. Service is instantiated;
2. Service is started, `startup()` sequence is executed. Note that the startup may be unsuccessful;
3. Service waits idly, allowing others to call its business logic specific methods;
4. Service is stopped, `shutdown()` sequence is executed.

A typical example of a passive service is `inetd`, a.k.a. superserver - when starting up, it reads the configuration, may fail if it is incorrect, then waits for incoming connections, serves them, then, on shutdown, it may kill all active connections, then it terminates.

The service is not a silver bullet. While a lot of consistency is added by enforcing the protection of `startup()` and `shutdown()` methods so they don't blow up and leave the service in an undefined state, there's a condition that must be watched carefully: the service that has just stopped must have a state identical to a service that has just been instantiated. In the `PassiveService` proper this condition is met, however, subclasses must watch the implementation closely. Usually, the best solution is to discard the instance of the service that has stopped and create a new one, however, it may not be feasible under some circumstances (poolable resources, high performance environment).

4. Active Service

Active service, unlike passive, doesn't sit idly after it has started. Instead, `execute()` is called at step #3. Also, the service may terminate by itself (calling `shutdown()`) if it so happens that `execute()` call is finished before some external entity called `stop()`.

A typical example of an active service is... a batch program. It starts, reads the configuration, does some predefined job and terminates either when it is done, or when the user kills it, closing the file handles as it is shutting down.

5. Using Semaphores with Services

In order to make the services useful, asynchronous notifications about the service state were introduced. There are four notification checkpoints (for both passive and active services):

1. When the service was started.
2. When the service completed the startup sequence, either successfully or unsuccessfully.
3. When the service was stopped.
4. When the service completed the shutdown sequence.

Here's an example illustrating the usefulness of the notification checkpoints:

```
protected void startup() throws Throwable {

  // Create a semaphore group to keep track on subservice startups
  SemaphoreGroup subStarted = new SemaphoreGroup();

  for ( ; needToStartSomeMoreSubServices(); ) {

    Service s = createNextSubService();

    // add the semaphore returned by start() to the group
    subStarted.add(s.start());
    // and keep going
  }

  // And now wait until all the subservices have started successfully
  if (!subStarted.waitForAll(true)) {
    throw new ServiceUnavailableException("Oops");
  }
}
```

In case there's a single service startup, the code is even simpler:

```
Service s = new Service();

if ( !s.start().waitFor() ) {
 throw new ServiceUnavailableException("Service startup failed");
}
```

Note:

The only caveat to watch for is that the service failure cause is not delivered to the entity that started the service. If that is necessary, use `Messenger` instead of the service, or use the Asynchronous Completion Token.