How to Connect Your Adaptation Engine to TAS?

In this document, we explain you how you can connect your adaptation engine to the TAS implementation. The prerequisite is that your engine can interact with Java code. We start with explaining the default behavior of TAS, i.e., without self-adaptation. We focus on service failures, but the approach is similar for other properties. Then we explain how probes and effectors work. Finally, we show how a very simple self-adaptation loop connects to TAS using probes and effectors to handle service failures.

Default behavior of TAS services.

When a user invokes the assistance service, the workflow engine invokes concrete services based on the specification of the workflow. A failure of a concrete service invocation is detected based on a timeout. The default timeout is three times the expected response time of a service as specified in the service description, but timeouts can be customized during configuration. Figure 1 shows the result of a series of invocations of the assistance service with the default behavior. The figure shows that whenever a concrete service fails, the composite service also fails.

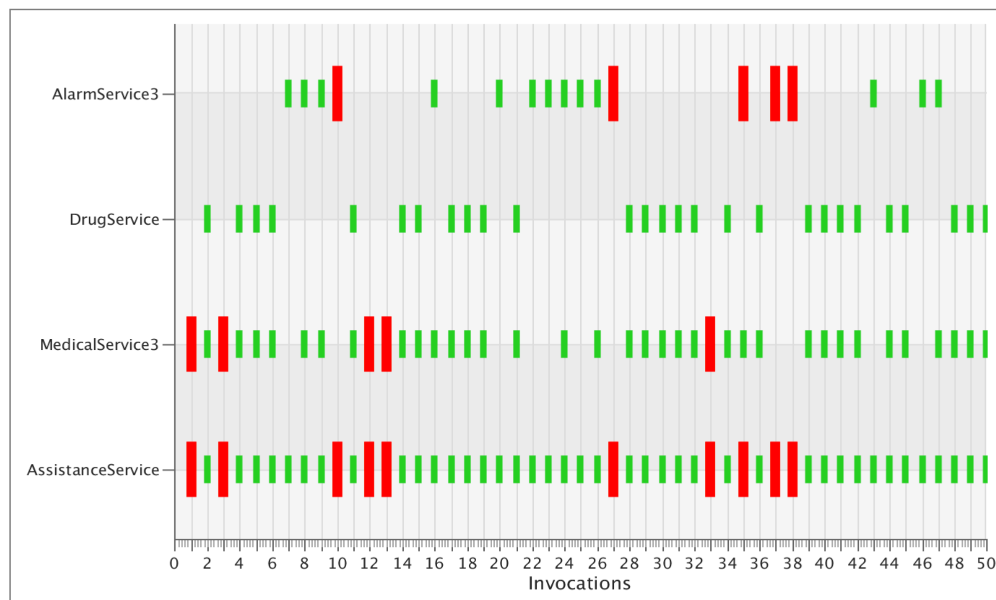

Figure 1. TAS invocation without adaptation

Probes and Effectors in TAS

ReSeP, the underlying service platform of the TAS implementation, provides probes and effectors. Probes and effectors enable assessing the status of the service system and effectors enable adapting the default behavior dynamically. ReSeP comes with a set of predefined probes and effectors, but this set can be extended as needed. Figure 2 shows a code excerpt of the `WorkflowProbeInterface` that supports monitoring workflow invocations (workflow started and workflow ended events) and invocations of concrete services (operation invoked, returned, service not found, and timeout events). Self-defined probes that implement `WorkflowProbeInterface` can register to the `WorkflowProbe` of the assistance service to receive event notifications. Whenever any of the six events happen, the probe will be notified by calling the corresponding method of the implementation.

```java
public interface WorkflowProbeInterface {
  /**
   *  Generates an event when the composite service starts a new execution of the workflow.
   */
  public void workflowStarted(String qosRequirement, Object[] params);

  /**
   *  Generates an event when a service operation is invoked by the workflow.
   */
  public void serviceOperationInvoked(ServiceDescription service, String opName, Object[] params);

  /**
   *  Generates an event when a service operation invoked by the workflow returns successfully.
   */
  public void serviceOperationReturned(ServiceDescription service, Object result, String opName, Object[] params);

  /**
   *  Generates an event when a service operation invoked by the workflow fails, that is, the service has not responded
   *   within a predefined timeout.
   */
  public void serviceOperationTimeout(ServiceDescription service, String opName, Object[] params);

  /**
   * Generates an event when the workflow cannot find a concrete service for the given service type and operation.
   */
  public void serviceNotFound(String serviceType, String opName);

  /**
   *  Generates an event when a workflow execution returns.
   */
  public void workflowEnded(Object result, String qosRequirement, Object[] params);
}
```

Figure 2 .Definition of the WorkflowProbeInterface

Figure 3 shows a code excerpt of the predefined `WorkflowEffector` that supports dynamic adaptation of the behavior of a workflow.

```java
public class WorkflowEffector extends AbstractEffector {
  /**
  * Update the workflow
  */
  public void updateWorkflow(String workflow) {
      …
  }

  /**
   * Remove the service from the list of available services
   */
  public void removeService(ServiceDescription sd){
      …
  }

  /**
   * Update the list of available services (from the service registry)
   */
  public void refreshAvailableServices(){
      …
  }

  /**
  * Update a service description in the list of available services
  */
  public void updateServiceDescription(ServiceDescription oldService, ServiceDescription newService) {
      …
  }
}
```

Figure 3 Class definition of the WorkflowEffector

The effector enables updating the workflow, removing services from the list of available services, updating custom field in the service descriptions of available services, and refreshing the list of available services.

Connecting Adaptation Engine to TAS with Probes and Effectors

We illustrate how a simple adaptation engine connects to TAS using a probe and an effector. The strategy of the adaptation engine consists of two parts: (1) when service fails it removes the service description of this failing service from the available services, and (2) when all service descriptions of a particular type of service are removed, the set of available is updated from the service registry.

We start with defining a probe to monitor the relevant events.

```
public class MyProbe implements WorkflowProbeInterface {
        MyAdaptationEngine myAdaptationEngine;
        public void connect(MyAdaptationEngine myAdaptationEngine) {
           this.myAdaptationEngine = myAdaptationEngine;
        }
        @Override
        public void serviceOperationTimeout(ServiceDescription service, String opName, Object[] params) {
           myAdaptationEngine.serviceFailure(service, opName);
        }
       @Override
        public void serviceNotFound(String serviceType, String opName){
           myAdaptationEngine.serviceNotFound(serviceType, opName);
        }
        // Other methods are not used and remain empty
        ……………………..
  }
```

Figure 4. Definition of MyProbe class

`MyProbe` implements `WorkflowProbeInterface` and tracks timeouts of service invocations and no services available of a particular type. The events are forwarded to the adaptation engine that is connected to the probe. For the effector, we use a predefined `WorkflowEffector.`

`MyAdaptationEngine` is then defined as follows:

```
public class MyAdaptationEngine {

        private WorkflowEffector effector;

        public MyAdaptationEngine(MyProbe probe, WorkflowEffector effector) {
                probe.connect(this);
                this.effector = effector;
        }

        public void serviceFailure(ServiceDescription service, String opName) {
              // Remove failed service from the available set of services
              effector.removeService(service);
        }

        public void serviceNotFound(String serviceType, String opName) {
              // Refresh set of available services
              effector.refreshAllServices(serviceType, opName);
        }
}
```

Figure 5. Definition of MyAdapter class

The adaptation engine connects with the probe and the effector. When the probe notifies a service failure, the engine removes the failing service from the set of

available services. When the probe detects that a service of a particular type is not found, the engine refreshes the set of available services for that type.

The following code snippet shows how the adaptation engine can be added to the TAS configuration:

```
// Add the simple adaptation to the TAS configuration
public class SimpleAdaptation implements TASConfiguration {

    MyProbe myProbe;
    WorkflowEffector myEffector;
    AssistanceService assistanceService;

    public SimpleAdaptation(AssistanceService assistanceService) {
            this.assistanceService = assistanceService;
            myProbe = new MyProbe();
            myEffector = new WorkflowEffector(assistanceService);
            MyAdaptationEngine myAdaptationEngine = new MyAdaptationEngine(myProbe, myEffector);
    }

    @Override
    public void setConfiguration() {
            assistanceService.getWorkflowProbe().register(myProbe);
    }
    @Override
    public void removeConfiguration() {
            assistanceService.getWorkflowProbe().unRegister(myProbe);
    }
}
```

Figure 6. Adding SimpleAdaptation to the TAS configuration

Figure 7, shows the result of using the adaptation engine with simple adaptation. The figure shows that when a service invocation fails, another instance of the same service type is invoked. There is only one case (invocation 18) where the assistance service fails when all services of the same type failed, i.e., MedicalAnalysisService.

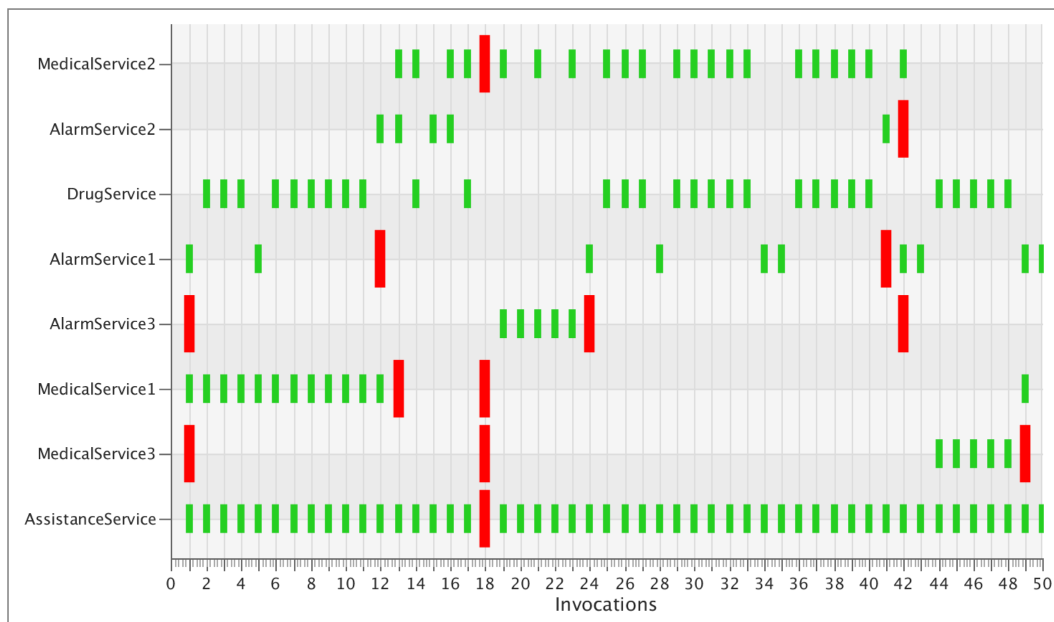

Figure 7. Run with simple adaptation