Linnéuniversitetet
Kalmar Växjö

Research Service Platform v.3.4

Author: M. Usman Iftikhar,
Yifan Ruan
Adptwise Group
Linnaeus University, Sweden
Email: usman.iftikhar@lnu.se

Contents

1 Introduction

Service oriented architecture (SOA) helps to define and use domain functions as services. We provide Research Service Platform (ReSeP), which is designed specifically for research purpose; to help students with creating service oriented applications in a convenient way.

Figure 1.1 shows the main components of ReSeP. ReSeP runs on top of Java runtime environment.

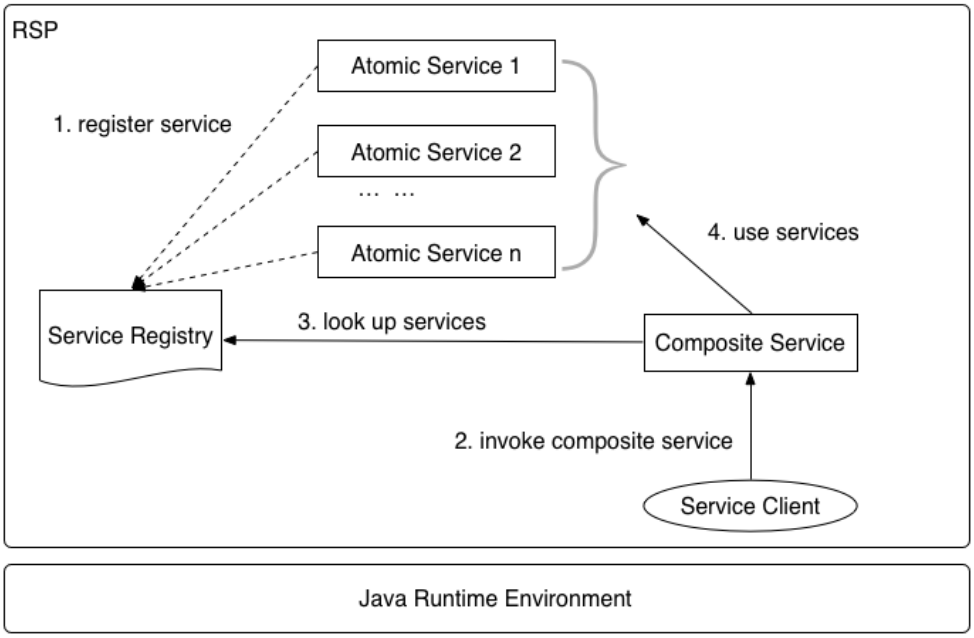

Figure 1.1 Overview of ReSeP

ReSeP supports two kinds of services, i.e., atomic services and composite services. Atomic services provide single domain functionality by means of operations that do not depend on other services. On the other hand, composite services are composed of other services based on a workflow. The workflow states how to compose other services. ReSeP provides a workflow engine, which is used by composite services to execute the workflow.

Service registry is a special kind of atomic service provided by ReSeP. It provides the functionality to register other services. Service registry also provides functionality to lookup other services. Services are registered to the service registry using a service description. A service description contains the name of the service, a list of operations supported by a service, an address, among other things.

ReSeP also provides support to use composite services, i.e., service clients. When a service client invokes a composite service, the composite service looks for services specified in the workflow through the service registry. The service registry receives service and operation names and

returns a list of service descriptions. The composite service uses that list to select services according to specified QoS requirements by the service client. Examples of the QoS requirements can be to select services depending upon minimum cost, minimum response time or both, maximum availability, etc.

2 Architecture Design

We use class diagrams and sequence diagrams to specify the architectural design of ReSeP.

2.1 Class diagram of ReSeP

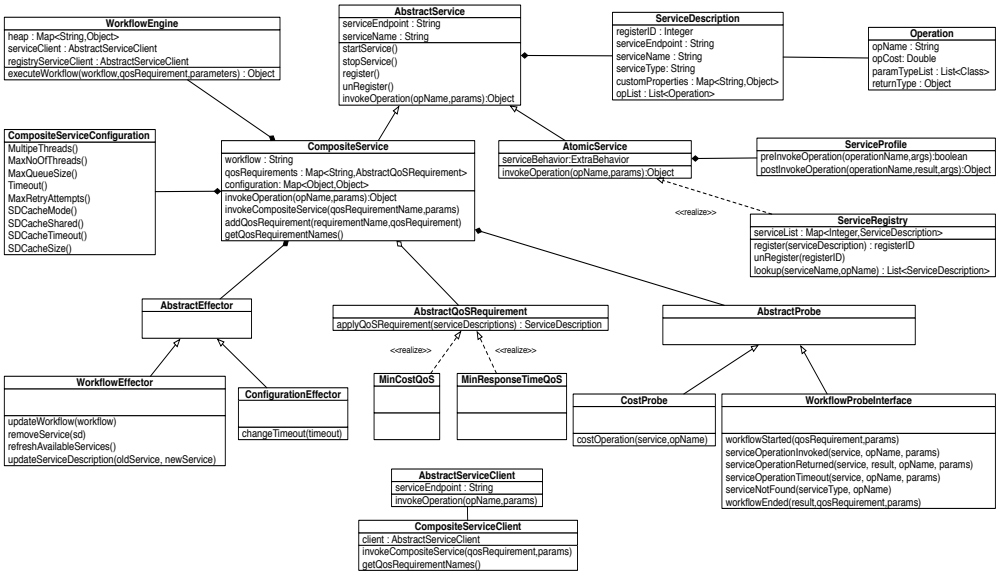

Figure 2.1 Class diagram of ReSeP

Figure 2.1 shows the class diagram of ReSeP. We explain each class in detail below. To provide understandability, we divide the classes into two sections. In the first section we discuss classes that are required to create a basic service-based system. In the second section, we discuss classes that can be used for experimentation purpose.

2.1.1 Basic service classes

AbstractService: is the core of ReSeP and a basic unit to create a service. `AbstractService` provides functionality to create atomic and composite services; it provides an abstraction of all the complex functionality, e.g., create a service, send and receive messages. Here is the list of methods provided by `AbstractService` class:

- startService(): starts the service. When this method is called, starts listening for incoming messages.

- stopService(): stops the service. When this method is called, the `AbstractService` stops the listening for incoming messages on the queue.
- register(): sends the service description to the service registry. Service registry returns a `registerID` indicating that this service is registered successfully.
- unRegister(): sends the `registerID` to the service registry to unregister the `AbstractService`.

AtomicService: inherits from `AbstractService` and provides a logical entity for creating atomic services.

ServiceRegistry: a repository to register/unregister services. `ServiceRegistry` also provides lookup functionality to search for service descriptions. There are three methods provided by service registry.

- register(): registers the actual services. This method saves the description of the service and returns a registration id.
- unRegister(): accepts the registration id and unregisters the service.
- lookup(): accepts two parameters, i.e., service name and operation name. When called, this method returns a list of services that have the same service name and operation name.

CompositeService: provides an abstraction to create composite services. `CompositeService` contains a workflow file that specifies a list of service operations to be invoked. The format of the workflow file is explained in section 2.3.1. It also contains a hash map of QoS requirements associated with their names, which are used to select a service among multiple service providers. Here is the list of methods provided by `CompositeService`.

- invokeCompositeService(): when this method is called, it use the workflow engine to execute the workflow specified in the workflow file. After executing, the workflow engine returns the result, which is sent to the service client.
- addQoSRequirement(): accepts an implementation of the `AbstractQoSRequirement` and a name for that requirement. This method saves the QoS requirements in a hash map. The name is used later on by service clients to select a QoS requirement.
- getQoSRequirementNames(): returns a list of all the names in the repository (hash map) of QoS requirements.

CompositeServiceConfiguration: set the configuration of composite service, which has nine properties:

- MultipeThreads: use single thread or multiple threads to deal with client requests. The default value is false, i.e., single thread only.

- MaxNoofThreads: the number of threads to be created when MultipeThreads is set to true.
- Timeout: maximum waiting time for response before a service is considered as failed. The default value 10 seconds.
- MaxRetryAttempts: maximum numbers of retry attempts before a service considered as failed. The default value is 1.
- SDCacheMode: use cache or not. If false, on each operation invocation, services will be retrieved from service registry. The default value is false. If this mode is true, then composite service will store a list of available services locally, and only retrieves services from service registry when needed.
- SDCacheShared: share the cache among multiple invocations.
- SDCacheTimeout: time interval to refresh cache automatically.
- SDCacheSize: max number of cache items stored in the cache

WorkflowEngine: used to execute the workflow of composite services. As mentioned before, a workflow consists of list of service invocations. To invoke a service, the engine looks up the service names via the lookup operation of the `ServiceRegistry`. The `ServiceRegistry` returns lists of corresponding service descriptions. After that the engine selects services according to the selected QoS requirement and subsequently invokes the services. `WorkflowEngine` has following method:

- executeWorkflow(): takes a workflow and QoS requirement for executing the workflow. When workflow is executed successfully, this method returns the result.

CompositeServiceClient: provides support for remotely accessing a composite service. It has two methods:

- invokeCompositeService(): invokes the composite service and returns the result. Using this method, we can also specify input parameters to the workflow and provide the QoS requirement to select by composite service.
- getQoSRequirementNames(): returns the list of QoS requirement names supported by the composite service.

AbstractQoSRequirement: is an interface to specify QoS requirements. Implementation of this interface is necessary to specify QoS requirements. Concrete examples of implementations of this interface are `MinCostQoS` and `MinResponseTimeQos`. `AbstractQoSRequirement` has only one method:

- applyQoSRequirement(): contains the logic to select a service from a list of service descriptions. Accepts three parameters, list of service description, operation name and operation parameters. The workflow engine calls this method automatically.

ServiceDescription: defines the format of a service description, It contains the following information:

- registerID: the unique id of service, obtained from the ServiceRegister after registration.
- serviceName: the name of service.
- serviceType: type of the service. It is calculated automatically from the class name.
- opList: all the operations supported by service. Information of each operation is gathered automatically by using Java reflection .
- serviceEndpoint: the endpoint, i.e., unique address of the service.
- customProperties: is a hash map to store some additional information about the service, e.g., cost, response time, etc. The classes, which implement `AbstractQoSRequirement`, can use this hash map to do selection among multiple services.

2.1.2 Experimentation classes

To enable researchers to experiment with the ReSeP platform, ReSeP offers three experimentation features. Here we discuss each feature with classes in detail.

1. Service Profile

Service profile is used to emulate the non-functional characteristics of the behaviour of services. For example, the profile may emulate a delayed response or a failure, e.g., to emulate periods with different workloads.
To create service profiles, ReSeP offers an abstract class, i.e.,`ServiceProfile`. The `ServiceProfile` class helps a service to define behaviour, before or after a service operation is invoked. This class consists of two methods. If needed any of this method can be overridden to change its intended behaviour.

- preInvokeOperation(): this method is called before any service operation is invoked of the atomic service. This method returns a Boolean value. If the value returned is true, the subsequent service operation will be invoked. Otherwise in case of false value, the service operation will not be invoked, and request to invoke service operation will be ignored.
- postInvokeOperation(): this method will be called after the service operation is invoked. One parameter of this method is result of the service operation, which was invoked. If needed, result can be modified as well. The result returned from this method will be sent to the composite service.

2. Probe

ReSeP offers probes that allow monitoring of particular events in the service system. There are two probe classes provided by ReSeP that are inherited from `AbstractProbe` class. Here we provide each probe in detail.

WorkflowProbe: helps to monitor the workflow execution. It has six methods:

- workflowStarted(): this method will be called before execution of the workflow engine is started.
- serviceOperationStarted(): during execution of the workflow, the composite service can invoke many atomic services. This method is called before invoking any of the atomic service operation.
- serviceOperationReturned(): this method is called when an atomic service operation is invoked successfully and returned the result.
- serviceOperationTimeout(): in case there is no response from the atomic service this method will be called. Atomic services can define a response time in their service description. 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.
- workflowEnded(): this method will be called after the workflow execution is finished.

CostProbe: allows calculating the cost of service invocations. This class has only one method:

- costOperation(): this method is called when a service operation is invoked successfully by composite service.

3. Effector

ReSeP offers effectors that enable runtime manipulation of the service application. Concretely, ReSeP provides two effectors that are inherited from `AbstractEffector` class.

WorflowEffector: allows to dynamically adapt behaviour of the workflow.

- updateWorkflow(): change the workflow to a new workflow. The composite service will execute the new workflow from the next invocations.
- removeService(): remove the service from the list of available services.

- refreshAvailableServices(): update the list of available services (from the service registry)
- updateServiceDescription():Update a service description in the list of available services
-

ConfigurationEffector: enables to update service configurations.

- changeTimeout(): change the timeout value of the composite service configuration.

2.2 Sequence diagrams

To help understanding the ReSeP we provide a number of sequence diagrams.

2.2.1 Register service

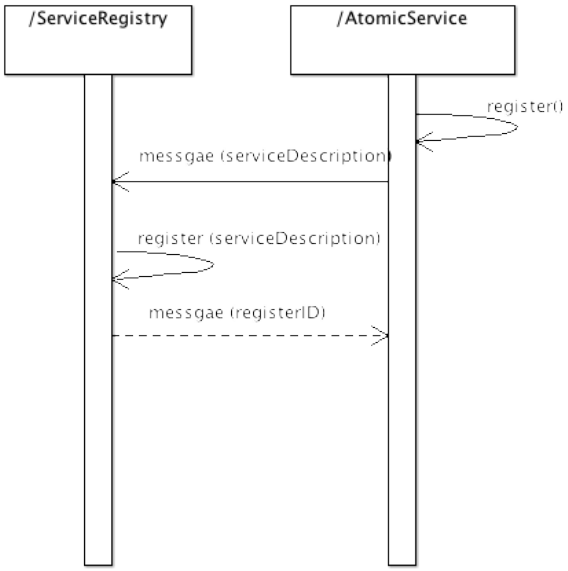

Figure 2.2.1 Sequence diagram of registering a service

Figure 2.2.1 presents the registration process between an `AtomicService` and `ServiceRegistery`. When the `register`() method is called by an `AtomicService`, the service description of the `AtomicService` is sent to the `ServiceRegistry`, which returns an unique register id.

2.2.2 Invoking a CompositeService

Figure 2.2.2 describes the process of invoking a `CompositeService` by a `CompositeServiceClient`. The `CompositeServiceClient` invokes the `CompositeService` remotely by sending a message. The `CompositeService` uses the workflow engine to execute the workflow and returns the result.

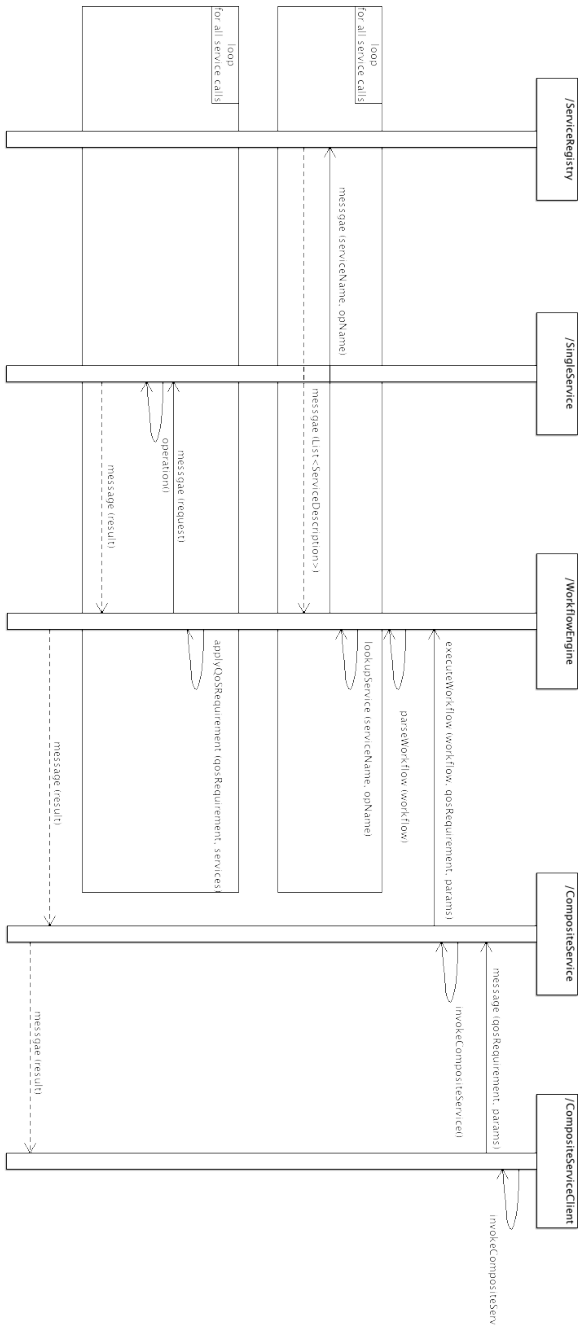

Figure 2.2.2 Sequence diagram of invoking a composite service

2.2.3 Workflow grammar

A workflow must begin with “start” tag and end with “return” tag, which contains a list of statements.
For example:

start [a, b, c]
Result1=Service1.Operation(a, b)
Result2=Service2.Operation(Result1, c)
return Result2

In the example, a, b, c are optional parameters to the workflow. A workflow may have a return value or not.

Grammar:
workflow := ‘start’ [parameters?] statement+

parameters := id (, id)*

block := ‘{’ statement* ‘}’ | statement

statement := forLoop | whileLoop | doWhileLoop | ifStatement | parallelStatement | returnStatement | expression

forloop := ‘for’ ‘(’ expression ‘;’ expression ‘;’ expression ‘)’ block

whileLoop := ‘while’ ‘(’ expression ‘)’ block

doWhileLoop := ‘do’ block ‘while’ ‘(’ expression ‘)’

ifStatement := ‘if’ ‘(’ expression ‘)’ block (‘else’ block)?

parallelStatement := ‘parallel’ block

returnStatement := ‘return’ (expression)?

expression:= assignment | methodInvocation | uniOperandInstruction | binaryOperandInstruction

methodInvocation := ID ‘.’ ID ‘(’expression (, expression)* ‘)’

uniOperandInstruction := uniOperator expression

binaryOperandInstruction:= expression binaryOperator expression

assignment := ID assignmentOperator expression

ID := (‘a’ .. ‘z’ | ‘A’ .. ‘Z’ | ‘_’) (‘a’ .. ‘z’ | ‘A’ .. ‘Z’ | ‘0’ .. ‘9’ | ‘_’)*

uniOperator := ‘not’ | ‘!’

binaryOperator := ‘+’ | ‘-’ | ‘*’ | ‘/’ | ‘%’ | ‘or’ | ‘and’ | ‘<<’ | ‘>>’ | ‘||’ | ‘&&’ | ‘>’ | ‘<’

assignmentOperator := ‘=’ | ‘*=’ | ‘/=’ | ‘%=’ | ‘+=’ | ‘-=’ | ‘<<=’ | ‘>>=’ | ‘&=’ | ‘^=’ | ‘|=’

Worth noting

- ✓ **Values**

We only support `Integer`, `Boolean` and `Null` values. And a variable can be defined by:

variable = 2
variable = [true/false]
variable = null

The variable not belongs to a special type, so we can assign any value to a variable. The name of a variable can be any except the reserved keyword: “start”, “return”, “parallel”, “if”, “else”, “do”, “while”, “true”, “false” and so on, most of which is similar as keywords in Java.

- ✓ **Parallel**

Statements inside the parallel block will be executed at the same time. And the workflow will be blocked until all the statements have been finished.

For example:

```
parallel {
    Service1.Operation(a, b)
    Result1=Service2.Operation(c)
    Result2=Service3.Operation(c)
}
```

Inside the parallel, corresponding operations of Service1, Service2, and Service3 will be executed simultaneously.

- ✓ **Operation**

For “Service.Operation()”, if “Service” is the name of one single service, then related operation will be invoked. And if “Service” is “this”, then local operation defined in composite service class will be invoked.

- ✓ **For statement**

Because a variable in workflow does not have a type, so we use “for statement” by:

```
for (i=0;i<5;i++){
        Service.Operation(a, b)
    }
```

The operation of service will be invoked five times automatically.

- ✓ **Comment**

We use “//” to comment one line and “/*,*/” to comment several lines, for example:

```
// comment1
/*
    comment2
    comment3
*/
```

- ✓ **We can treat the rest of workflow grammar as Java grammar**

3 How to use ReSeP

In this section, we describe how to use the ReSeP. We describe the steps with the help of one example Tele Assistance system. We start with introducing the example in the following section and then we discuss in detail how we can use ReSeP to implement the example.

3.1 Tele Assistance System

The Tele Assistance System (TAS) is a home device that offers health support to patients. The system comprises a composite service that uses the following atomic services:

- Alarm Service, which provides the operation `sendAlarm`.
- Medical Analysis Service, which provides the operation `analyzeData`.
- Drug Service, which provides the operations `changeDoses` and `changeDrug`.
- The TAS workflow, shown in the figure 3.1, starts executing as soon as a user enables the home device supplied by the TAS provider. The workflow then enters an infinite loop whose iterations start with a "pick" activity that suspends the execution and waits for one of the following three messages: 1) `vitalParamsMsg`, 2) `pButtonMsg`, or 3) `stopMsg`.

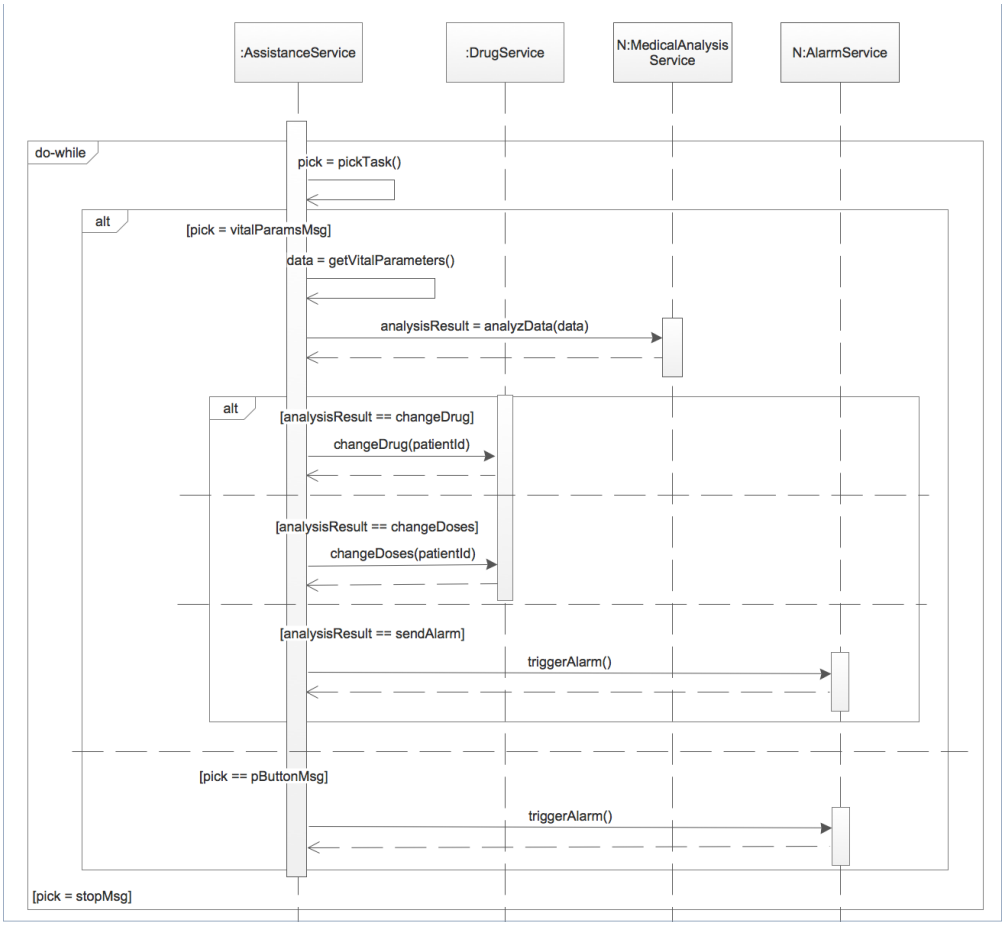

Figure 3.1 Sequence diagram of Tele Assistance System

The first message contains the patient's vital parameters, which are forwarded by the workflow engine to the Medical Analysis Service (MAS) by invoking the operation `analyzeData`. The MAS is in charge of analyzing the data, and replies by sending a result value stored in a variable `analysisResult` that contains a value that can be `changeDrug`, `changeDoses`, or `sendAlarm`. A `changeDrug` or `changeDoses` value triggers a Drug Service, which runs on a home device, adapting the medication of the patient accordingly. A `sendAlarm` value triggers an Alarm Service that will trigger the intervention of a First-Aid Squad (FAS) comprised of doctors, nurses, and paramedics whose task is to visit the patient at home in case of emergency. The message `pButtonMsg` caused by pressing a panic button also generates an alarm sent to the FAS. Finally, the message `stopMsg` indicates that the patient decided to cancel the TAS service, deleting each pending invocation to the FAS service.

Different providers could be involved in providing concrete implementations for the atomic services in the TAS. Concretely, we consider that 3 and 5 telecommunication operators implemented the Alarm Service

and the Medical Analysis Service respectively, each such concrete service being provided with different cost, performance and reliability characteristics. Finally, we consider that a single implementation of the Drug Service is available. The following table gives an overview of the declared characteristics of the different concrete services.

Service	Name	Failure Rate	Execution Time
S11	Alarm Service 1	0.03	1.1
S12	Alarm Service 2	0.04	0.9
S13	Alarm Service 3	0.01	0.3
S21	Medical Analysis Service 1	0.01	2.2
S22	Medical Analysis Service 2	0.01	2.7
S23	Medical Analysis Service 3	0.02	3.1
S24	Medical Analysis Service 4	0.03	2.9
S25	Medical Analysis Service 5	0.05	2.0
S31	Drug Service 1	0.02	

3.2 Implementation

In this section we discuss in detail about how TAS can be implemented using ReSeP. We divided the implementation in two steps. In the first step, we implement TAS using basic service classes. In the second step, we use the implementation features of ReSeP to emulate TAS.

Note: Before using the following steps, add the ReSeP.jar to your project folder build path with all the jar files from the provided required-libs folder.

3.2.1 Using basic service classes

In this section, we discuss how to use basic service classes to implement TAS.

Create workflow

Composition helps to use other services. Usually composition is done with creating a workflow of the system. In ReSeP, we can create a workflow using syntax provided in section 2.2.3. Here we provide workflow of our example Tele Assistance System.

```
START [patientId]

        // Declaration of button constants
        vitalParamsMsg = 1
```

```
        pButtonMsg = 2

        // Declaration of analysis constants
        changeDrug = 1
        changeDoses = 2
        sendAlarm = 3

        pick = this.pickTask()
        if (pick == vitalParamsMsg){

                data = this.getVitalParameters()
                analysisResult = MedicalService.analyzeData(data)

                if (analysisResult == changeDrug)
                        DrugService.changeDrug(patientId)
                else if (analysisResult == changeDoses)
                        DrugService.changeDoses(patientId)
                else if (analysisResult == sendAlarm)
                        AlarmService.triggerAlarm(patientId)
        }
        else if (pick == pButtonMsg) {
                AlarmService.triggerAlarm(patientId)
        }

RETURN
```

Run ServiceRegistry

ServiceRegistry contains a main method to run the service registry from a console. If you want to use or run the service registry through code you can apply the following lines of code:

```
public static void main(String[] args) {
    ServiceRegistry serviceRegistry=new ServiceRegistry();
    serviceRegistry.startService();
}
```

Create atomic service

A class can be created as atomic service by extending from the AtomicService class. A service can have multiple service operations. Service operations are the operations that are accessible to composite services. To create a service operation, we must annotate a method of the service class by "*@ServiceOperation*" annotation. Using the annotation, the underlying platform will automatically find the service operations and invoke them when called (using Java reflection). We illustrate the creation of an atomic service with the help of medical analysis service:

```
public class MedicalAnalysisService extends AtomicService {

        private int medicalServiceId;

        public MedicalAnalysisService(String serviceName, String serviceEndpoint, int medicalServiceId) {
                super(serviceName, serviceEndpoint);
                setMedicalServiceId(medicalServiceId);
        }
```

```
	@ServiceOperation
	public int analyzeData(HashMap data){
		System.out.println("Medical service " + medicalServiceId + " is started to analyze the data.");

		 // Logic here to perform medical analysis

		return analysisResult;
	}

	public int getMedicalServiceId() {
		return medicalServiceId;
	}

	public void setMedicalServiceId(int medicalServiceId) {
		this.medicalServiceId = medicalServiceId;
	}
}
```

After creating the atomic service, next steps are to instantiate the atomic service, add declared characteristics and register to the service registry. Below is the code snippet, which is used to perform these steps.

```
MedicalAnalysisService medicalAnalysis1 = new MedicalAnalysisService("MedicalService","service.medical1",1);
medicalAnalysis1.getServiceDescription().setResponseTime(2200);
medicalAnalysis1.getServiceDescription().getCustomProperties().put("FailureRate", 0.01);
medicalAnalysis1.setServiceBehavior(new AtomicServiceFailureBehavior(1));
medicalAnalysis1.startService();
medicalAnalysis1.register();
```

There can be many instances of the atomic services, and each instance should have the same name, e.g., we use the name "*MedicalService*" for all the instances of the medical analysis service. Opposite to the service name, all the atomic services should have a unique service endpoint. In our example above "*service.medical1*" is the unique endpoint, where only medical service 1 will listen for the requests.

Properties of the services can be added into the service description, e.g., *FailureRate*. A service starts listening for messages at endpoint using "*startService*" method and registers to the registry by "*register*" method.

Create composite service

A composite service can be created by extending the CompositeService class. A composite service class can also define service operations and local operations. Local operations are defined by annotating "*@localOperation*" before methods. Workflow engine invokes automatically local operations when the local operations are called by "this.Operation()" syntax. Here we create a composite service from our example TAS, i.e., Assistance service, to illustrate how to create a composite service and define local operations.

```
public class AssistanceService extends CompositeService {

	public AssistanceService(String serviceName, String serviceEndpoint,
			String workflowFilePath) {
		super(serviceName, serviceEndpoint, workflowFilePath);
```

```
	}

	@LocalOperation
	public int pickTask() {

		System.out.println("Pick (1) to measure vital parameters, (2) for emergency and (3) to stop
		service.");
		Scanner in = new Scanner(System.in);
		do {
			String line = in.nextLine();
			Integer pick = Integer.parseInt(line);
			if (pick < 1 || pick > 3) {
				System.err.println("Wrong value:" + pick);
			} else {
				return pick;
			}
		} while (true);
	}

	@LocalOperation
	public HashMap getVitalParameters(){
		return calculateVitalParameters();
	}
}
```

There are two local operations defined in the above example, i.e., "*pickTask()*" and "*getVitalParameters()*". The "*pickTask*()" operation is used to take commands from the patient. The "*getVitalParameters()*" returns the vital parameters of the patient. The Composite service classes can be instantiated as the same way we instantiate atomic services. One particular difference is that in the constructor of composite service we need to provide path of the workflow as well. Following code provides the snippet to instantiate the assistance service.

```
AssistanceService assistanceService
				= new AssistanceService("TeleAssistanceService", "service.assistance", workflowPath);
assistanceService.startService();
assistanceService.register();
```

CompositeServiceConfiguration

CompositeServiceConfiguration is an annotation class that can be used to configure the behaviour of a composite service. Following is an example to describe how to use this composite service configuration:

```
@CompositeServiceConfiguration(
		Timeout = 20,
		SDCacheMode=true
)
public class  AssistanceService extends CompositeService{
	……..
}
```

In the above example, the CompositeServiceConfiguration is used to set maximum timeout to wait for a service invocation, and turn on the cache mode. There are more configurations available in that are can be configured similar as the above example.

Specify QoS requirements

Multiple vendors can provide multiple instances of an atomic service. For example in TAS, there are multiple instances of the alarm service and medical analysis services implemented by the tele communication operators.
A QoS requirement helps to choose an atomic service among the multiple instances of that atomic service provided through some QoS requirements. In TAS, we have a QoS requirement to provide reliability by selecting services based on minimum failure rate. Following is the QoS requirement specified for TAS.

```
public class QoSReliability implements AbstractQoSRequirement{

  @Override
  public ServiceDescription applyQoSRequirement(List<ServiceDescription> serviceDescriptions, String opName, Object[]params)
  {
        int minFailureRate = Integer.MAX_VALUE;
        int index = 0;
        int cost;
        HashMap properties;
        for (int i = 0; i < serviceDescriptions.size(); i++) {
          properties = serviceDescriptions.get(i).getCustomProperties();
          if (properties.containsKey("FailureRate")){
                cost = (int) properties.get("FailureRate");
                if (cost < minFailureRate){
                  minFailureRate = cost;
                  index = i;
                }
          }
        }

        return serviceDescriptions.get(index);
  }
}
```

The interface "*AbstractQoSRequirement*" is used to defined a QoS requirement. This interface has only one method "*applyQoSRequirement*", which is called when the composite service get multiple instances of an atomic service from registry or cache. Multiple QoS requirements can be registered with the composite service. Following is the code snippet to register the above QoS requirement with the assistance service.

```
assistanceService.addQosRequirement("QoSReliability", new QoSReliability())
```

Create composite service client

The following example shows how to create a composite service client that invokes a composite service.

```
CompositeServiceClient client = new CompositeServiceClient("service.assistance");
client.invokeCompositeService("QoSReliability", patientId);
```

The composite service endpoint is required to instantiate the composite service client. In the above example, the end point of the assistance service is provided in the constructor. The client has one function "*invokeCompositeService*" which invokes the composite service with following parameters: QoS requirement to use, and varargs list of parameters for the workflow.

Following is the code snippet to get list of all the QoS requirements names provided by the composite service.

```
List<String> qosRequirements = client.getQosRequirementNames();
```

3.2.2 Using experimentation features

In this section, how to use experimentation features of ReSeP to emulate TAS. 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.

Add service profile

ReSeP provides support to attach service profile to an atomic service. A service profile specifies behaviour is added before and after invocations of service operations. For example, in TAS we use a service profile to emulate service failures according to the defined failure rates.

```
public class AtomicServiceProfile extends ServiceProfile {

	private float failureRate;   // failure ratio of 100 invocations

	public AtomicServiceFailureBehavior(float failureRate){
		this.failureRate=failureRate;
	}

	@Override
	public boolean preInvokeOperation(String operationName, Object... args) {
		Random rand=new Random();
		if((rand.nextFloat())<=failureRate){
			return false;
		}
		return true;
	}

	@Override
	public Object postInvokeOperation(String operationName, Object result,
			Object... args) {
		return result;
	}
```

}

Create probes & effectors

Probes help to assess the system state. In ReSeP, a probe can be attached to a composite service to monitor execution of the workflow, invocation of services, and calculating the cost of invocations. There are two type of probes provided by ReSeP, i.e., `WorkflowProbe`, `CostProbe`. The `WorkflowProbe` helps to monitor workflow invocations (workflow started and workflow ended events) and invocations of concrete services (operation invoked, returned, service not found, and timeout events). The `CostProbe` helps to determine cost of each successful service invocation. Following is the code snippet to use `WorkflowProbeInterface`.

```
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
          ……..
  }
```

`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);
            }
}
```

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 probe can be registered with composite service and effector can be instantiated.

```
MyProbe myProbe = new MyProbe();
assistanceService.getWorkflowProbe().register(myProbe);
WorkflowEffector myEffector = new WorkflowEffector(assistanceService);
MyAdaptationEngine myAdaptationEngine = new MyAdaptationEngine(myProbe, myEffector);
```