

2. Distributed Services

Description

A "distributed service" is a service running on multiple peers within a given Service Space. They have a name which uniquely identifies them, once again within a given Service Space, and can be remotely invoked via a RPCish API.

Implementation & Registration of a Distributed Service

Any class implementing a set of interfaces is a valid distributed service implementation. Implemented interfaces capture the contracts, i.e. methods, which can be remotely executed.

Optionally, a service may implement the `org.codehaus.wadi.core.Lifecycle` interface. If implemented, it receives lifecycle callbacks when it is started or stopped. A service is started or stopped by the service registry it has been registered with when the associated service space starts or stops.

The following snippets show a basic service space implementation:

```
import java.util.concurrent.Executor;
import org.codehaus.wadi.core.Lifecycle;

public class BasicTaskExecutor implements
Executor, Lifecycle {
    public static final ServiceName NAME =
new ServiceName("Executor");

    public void start() throws Exception {
    }

    public void stop() throws Exception {
    }

    public void execute(Runnable command) {
        command.run();
    }
}
```

and its registration:

```
ServiceRegistry serviceRegistry =  
serviceSpace.getServiceRegistry();  
serviceRegistry.register(BasicTaskExecutor.N  
AME, new BasicTaskExecutor());
```

Remote Invocation

A service defined by a given service space can be remotely executed via a RPC like API. This API offers multiple configuration parameters developers can use to specify how invocations are to be performed.

The first step is to retrieve a ServiceProxyFactory from the service space defining the service to be invoked:

```
ServiceProxyFactory proxyFactory =  
serviceSpace.getServiceProxyFactory(BasicTas  
kExecutor.NAME, new Class[] { Executor.class  
});
```

The first parameter of getServiceProxyFactory is the name of the targeted service and the second is an array of interfaces the service implements.

The returned ServiceProxyFactory allows the configuration of global invocation parameters, which are inherited by all the service proxies created by this factory. For instance, here is how to set a global request-reply timeout to 2000ms:

```
InvocationMetaData invocationMetaData =  
proxyFactory.getInvocationMetaData();  
invocationMetaData.setTimeout(2000);
```

The actual creation of a service proxy is done through the getProxy() contract. The returned object is a ServiceProxy instance, which also implements the interfaces passed in to getServiceProxyFactory. Invocation parameters can be configured for this instance by retrieving the attached InvocationMetaData and the remote invocation is simply done by casting the instance to one of the service interfaces and executing it. For instance, this is how our BasicTaskExecutor service is executed on peer1:

```
ServiceProxy proxy =
proxyFactory.getProxy();
proxy.getInvocationMetaData().setTargets(new
Peer[] {peer1});
Executor executor = (Executor) proxy;
executor.execute(runnable); // note that
runnable must be serializable
```

Controlling Remote Invocations

The way remote invocations are executed is controlled by the `InvocationMetaData` instance attached to `ServiceProxyFactory`, for inherited global set-up, and `ServiceProxy`, for instance specific set-up.

The following invocation parameters are available:

- `timeout`: number of milliseconds a service proxy waits for a response. If a response is not received before the configured value, then the service proxy throws a `ServiceInvocationException` exception.
- `oneWay`: boolean indicating that the invocation is a one-way one. This means that an invocation request is sent, however no response is expected.
- `targets`: array of `Peers` to which invocation requests are dispatched. By default, invocations are done against all the `Peers` hosting the distributed service.
- `replyAccessor`: the result of a service invocation on a given `Peer` is usually sent back to the invoking `Peer`. In specific scenario, such a behavior is inadequate meaning that the result should not be propagated back. The `ReplyRequiredAssessor` strategy implementation is used to provide such a flexibility and prevent results to be returned to the invoking `Peer` depending on the service invocation result.
- `invocationResultCombiner`: when an invocation is performed against multiple `Peers`, the returned values of each service invocation performed on each targeted `Peer` need to be combined. The `InvocationResultCombiner` strategy implementation is used to combine multiple returned values into a single one.