QakService24WithInteraction

Progetto servicemath24Usage

Oltre i singoli protocolli

Costruiamo un client che interagisce con il servizio introdotto in *QakService24Usage* in modo diverso da quanto visto in quella sede, lasciando sullo sfondo i dettagli del protocollo usato.

L'obiettivo è rendere il **progetto del client e il suo codice** focalizzato sulla logica applicativa: e sul tipo sincrono/asincrono di comunicazione che si vuole realizzare.

In altre parole, facciamo un primo passo verso una direzione che pone in primo piano il **COSA fare**, piuttosto che sul **COME fare**.

Ma … il diavolo si nasconde nei dettagli !

Svantaggi nell'uso di astrazioni

Il passaggio dal *come* al *cosa* significa che molti aspetti, anche rilevanti, sull'uso dei protocolli verranno inseriti nella *parte sommersa* del client.

Con questo modo di procedere si **perderà il controllo** di molti particolari, che potrebbero rilverarsi importanti.

Vantaggi con l'uso di astrazioni

Avremo però il vantaggio di maggiore astrazione ed economia concettuale, utile nelle prime fasi fasi del processo di sviluppo software: l'**analisi** dei requisiti e del problema e l'impostazione di una **prima progettazione dell'archiettura** del sistema. (Si veda la domanda *ChatGPT Ingegneria del software*).

Su questo punto focalizzeremo l'attenzione di buona parte delle nostre attività future.

Dai protocolli a Interaction

La libreria *unibo.basicomm23* fornisce supporti che:

- introducono un concetto (**astrazione**) di alto livello: quello di *Interconnessione* punto-punto tra due componenti software
- definiscono l'interfaccia *Interaction* come contratto che gli oggetti che dovranno realizzare l'astrazione dovranno rispettare
- realizzano utility per l'uso di vari *protocolli*, implementando l'interfaccia *Interaction* per alcuni di essi (**tcp**, **udp**, **coap**, **mqtt**, **ws**, **http**).
- forniscono la classe *unibo.basicomm23.utils.ConnectionFactory* come factory degli oggetti-supporto che realizzano l'astrazione *Interconnessione*.

Lo schema del codice di un client può ora essere schematizzato come segue:

Creazione oggetto di supporto

1. Definizione del *protocollo* da usare
2. Definizione del messaggio di richiesta
3. Specifica (protocol-related) dell'host del servizio
4. Specifica (protocol-related) dell'accesso al servizio
5. Creazione mediante *ConnectionFactory* di un oggetto che implementa *Interaction*

```
/*1*/ProtocolType protocol = ProtocolType. ...;
/*2*/IApplMessage req = ....;
/*3*/String hostAddr = "...";
/*4*/String entry    = "..."

/*5*/Interaction conn =
      ConnectionFactory.createClientSupport(
                    protocol, hostAddr, entry);
```

Richiesta sincrona

1. Il metodo **request** blocca il client fino alla ricezione della risposta.

```
/*1*/ String answer =
           conn.request(req.toString());
```

Richiesta asincrona

1. Il metodo **forward** è di tipo fire-and-forget
2. Il metodo **receiveMsg** blocca il client fino alla ricezione della risposta.

```
/*1*/ conn.forward(req.toString());
...
/*2*/ String answer = conn.receiveMsg();
```

Tra le due chiamate, il client può eseguire altre azioni.

Riscriviamo dunque quanto fatto in *QakService24Usage*.

ServiceCallerInteraction

La impostazione del client è, come al solito, relativa alla definizione di un metodo **doJob**, cui affidiamo, in questo caso, il compito di usare tutti i protocolli.

ServiceCallerInteraction: impostazione

1. Dichiarazione del supporto che realizza *Interaction*
2. Definizione del messaggio di richiesta
3. Definizione della topic MQTT usata per la risposta
4. Invio della richiesta usando TCP
5. Invio della richiesta usando MQTT
6. Invio della richiesta usando CoAP
7. Invio della richiesta usando WebSocket
8. Attesa, per non perdere possibili ricezioni asincrone delle risposte
9. Terminazione del client

```
package main;
import unibo.basicomm23.interfaces.IApplMessage;
import unibo.basicomm23.interfaces.Interaction;
import unibo.basicomm23.mqtt.MqttConnection;
import unibo.basicomm23.msg.ProtocolType;
import unibo.basicomm23.utils.BasicMsgUtil;
import unibo.basicomm23.utils.CommUtils;
import unibo.basicomm23.utils.ConnectionFactory;

public class ServiceCallerInteraction {
/*1*/ private Interaction conn ;
      private String  nfibo = "21";
      private String payload="dofibo(N)".replace("N", nfibo);
/*2*/  private IApplMessage req = BasicMsgUtil.buildRequest(
        "clientJava", "dofibo", payload,"servicemath");
/*3*/ private String mqttAnswerTopic = "answ_dofibo_clientJava";

 public void doJob() {
  try {
/*4*/  selectAndSend(ProtocolType.tcp);
/*5*/  selectAndSend(ProtocolType.mqtt);
/*6*/  selectAndSend(ProtocolType.coap);
/*7*/  selectAndSend(ProtocolType.ws);
/*8*/  Thread.sleep(5000);
/*9*/  System.exit(0);
  }catch(Exception e){
      CommUtils.outred("ERROR " + e.getMessage() );
  }
 }

/*10*/protected void selectAndSend(
    ProtocolType protocol) throws Exception{
    ...
}
...
 public static void main( String[] args) {
      new ServiceCallerInteraction().doJob();
 }
}
```

10. Specifica del metodo di invio richiesta

ServiceCallerInteraction: invio richiesta

Invio della richiesta

1. Impostazione parametri per TCP
2. Impostazione parametri per CoAP
3. Impostazione parametri per MQTT
4. Impostazione parametri per WS
5. Creazione supporto
6. Possibile settaggio per tracing
7. Invio richiesta in modo **sincrono**
8. Invio richiesta in modo **asincrono**
9. Chiusura della connessione

```
protected void selectAndSend(
        ProtocolType protocol) throws Exception{
    String hostAddr="";
    String entry   ="";
    switch( protocol ) {
/*1*/ case tcp : {
        hostAddr = "localhost";
        entry    = "8011";
        break;
      }
/*2*/  case coap : {
        Connection.trace = true;
        hostAddr = "localhost:8011";
        entry    = "ctxservice/servicemath";
        break;
      }
/*3*/ case mqtt : {
        hostAddr = "tcp://broker.hivemq.com";
        entry    = "unibo/qak/servicemath";
         break;
      }
/*4*/ case ws : {
        hostAddr = "localhost:8088/accessgui";
        entry    = "request";
        break;
      }
      default:{
      }
    }//switch
/*5*/ conn = ConnectionFactory.createClientSupport(
                          protocol, hostAddr, entry);
/*6*/  //((Connection)conn).trace = false;
/*7*/  sendRequestSynch( req, conn, protocol );
/*8*/  sendRequestAsynch( req, conn, protocol );
/*9*/ conn.close();
}
```

ServiceCallerInteraction: richiesta sincrona

Richiesta sincrona

1. La richiesta in caso di WS è il numero
2. In ogni altro caso è un IApplMessage

```
protected  void sendRequestSynch(
   IApplMessage m,
   Interaction conn,ProtocolType protocol) throws Exception{
  String answer = "todo";
/*1*/if(protocol==ProtocolType.ws)
          answer = conn.request(nfibo);
     else
/*2*/answer = conn.request(req.toString());
     CommUtils.outmagenta(
        protocol+" | sendRequestSynch answer="+answer);
}
```

ServiceCallerInteraction: richiesta asincrona

Richiesta asincrona

1. Nel caso di MQTT, preparo un oggetto che riceve la risposta
2. La richiesta in caso di WS è il numero
3. In ogni altro caso, la richiesta è un IApplMessage
4. Attesa della risposta

```
protected  void sendRequestAsynch(
  IApplMessage m,
  Interaction conn,ProtocolType protocol)
                                  throws Exception{
if(protocol==ProtocolType.mqtt)
/*1*/ ((MqttConnection) conn).
       setupConnectionForAnswer(mqttAnswerTopic);
if(protocol==ProtocolType.ws)
/*2*/  conn.forward( nfibo );
else
/*3*/  conn.forward(req.toString());
/*4*/String answer = conn.receiveMsg();
CommUtils.outmagenta(
  protocol + " | sendRequestAsynch answer=" + answer);
}
```

Tra le due chiamate, il client può eseguire altre azioni.

NOTA: Va notato (e approfondito) come, nel caso **MQTT** e **WS**, l'oggetto che realizza la connessione sia capace anche di predisporre un oggetto per gestire le risposte inviate dal servizio.

Il caso HTTP

La richiesta viene inviata via POST e viene gestita lato service da un **M2MController**, appositamente introdotto nel service allo scopo di fare un esempio di interazione **M2M** con **HTTP**.

Richiesta M2M HTTP

1. URI della risorsa gestito da **M2MController**
2. Payload da inviare via POST
3. Oggetto per la connessione
4. Invio della richiesta via POST e attesa della risposta da parte di **M2MController**

```
protected void m2mHTTP() {
 try {
 /*1*/ String hostAddr="localhost:8088/RestApi/testHTTP";
 /*2*/ String entry   = req.toString();
 /*3*/ conn = ConnectionFactory.createClientSupport(
                        ProtocolType.http, hostAddr, "");
 /*4*/ String answer = conn.request(entry);
  CommUtils.outmagenta(
     "ServiceCallerInteraction | useHTTP answer=" + answer);
}catch(Exception e){
   CommUtils.outred("ERROR " + e.getMessage() );
 }
}
```

(che riceve la risposta stessa dal service applicativo)

Richiesta HMI HTTP

1. URI della risorsa gestito da **M2MController**
2. Payload da inviare via POST
3. Oggetto per la connessione
4. Invio della richiesta via POST e attesa della risposta da parte di **M2MController** (che riceve la risposta stessa dal service applicativo)

```
protected void hmiHTTP() {
 try {
 /*1*/ String hostAddr="localhost:8088";
 /*2*/ String entry   = "/";
 /*3*/ conn = ConnectionFactory.createClientSupport(
                        ProtocolType.http, hostAddr, "");
 /*4*/ String answer = conn.callHttp(entry);
  CommUtils.outmagenta(
     "ServiceCallerInteraction | useHTTP answer=" + answer);
 }catch(Exception e){
    CommUtils.outred("ERROR " + e.getMessage() );
 }
}
```

NEXT–> : a questo punto è utile mettere alla prova quanto affermato in *Vantaggi con l'uso di astrazioni* affrontando il progetto e la realizzazione di un *Sistema Produttore-Consumantore*.