3.2.13 D- US-13 - Simple Client Server

3.2.13.1

Editorial note
☐

Submitted by: Enigmatec Corporation Ltd. [URL]

3.2.13.2 Actors

Client - Role that makes requests to a Server.

Server - Role that responds to requests from a Client.

3.2.13.3 Description

A system is to be constructed from two Web Services that we call a "Client" and a "Server". The Client starts a session with a Server and then sends any number of requests receiving, in each instance, a response, and then closes that session with the Server. The Server has analagous behaviour.

3.2.13.4 Preconditions

The client has declared it's external observable behaviour as described above.

The server has declared it's external observable behaviour as described above.

3.2.13.5 Triggering Event(s)

The triggering event for the choreography between the Client and Server takes place when and only when an "open" request is made to the "Server".

3.2.13.6 Postconditions

There are not explicit post-conditions. Implicitly the system must return to Idle for both the Client and the Server and this can be done by ensuring that the Client sends a "close" and that the Server is ready to receive a "close"

3.2.13.7 Flow of Events

3.2.13.7.1 Basic Flow (Primary Scenario)

In it's initial state the Server is said to be Idle, we call this an "IdleServer". An IdleServer can only accept "open" requests. No other request is accepted by an IdleServer. When an IdleServer accepts an "open" request it's behaviour changes. We refer to this change as the Server behaving like a "BusyServer".

In it's BusyServer state a BusyServer can accept a "request" and then sends a response. Or it can also accept a "close" in which case it behaves like an IdleServer once more.

In pi-calculus we represent this as:

IdleServer(o,req,resp,c) = o-in.BusyServer(o,req,resp,c)

BusyServer(o,req,resp,c) = req-in.resp-out.BusyServer(o,req,resp,c) +
c-in.IdleServer(o,req,resp,c)

In it's initial state the Client is also said to be an "IdleClient" until it sends an open request. It then is said to be a "BusyClient".

As a "BusyClient" it can send a request and then accept a response. Or it can send a close, in which case it behaves like an IdleClient.

IdleClient(o,req,resp,c) = o-out.BusyClient(o,req,resp,c)

BusyClient(o,req,resp,c) = req-out.resp-in.BusyClientr(o,req,resp,c) +
c-out.IdleClient(o,req,resp,c)

3.2.13.7.2 Alternate Flow(s)

None defined.

3.2.13.8 Related Use Cases

None defined.

3.2.13.9 Requirements

Each process (Client and Server) has sequential behaviour that is visible as well as decision points (choices or the + operator in pi-calculus). In order to express the behaviour of the individual processes we require:

1. The ability to uniquely name a behaviour (i.e. IdleServer, BusyServer, IdleClient, BusyClient).
2. The ability to describe a behaviour in terms of their interaction. Their interaction is defined as the sending of a communication on a named channel (o,req,resp,c).

3. The ability to describe a behaviour recursively (i.e. the defn of BusyServer refers to itself).
4. The ability to describe a behaviour that has choice to enable different behaviours to be references (i.e. the use of the + operator).
5. The ability to describe a sequence of communications (i.e. the use of the "." operator in req-out.resp-in as well as o-in.BusyServer).

The composition, the behavioural contract that exists between a Client and a Server can be defined in pi-calculus as follows:

ComposedSystem = (!IdleClient | IdleServer)

The Composed system enables Clients to communicate with a Server. IdleClient1 communicates with the IdleServer until it sends a close. When a close has been received the transitions from behaving like a BusyServer to an IdleServer and so is able to accept open requests from any IdleClienti.

1. This ability compose behaviours into higher level processes such as our ComposeSystem above gives rise to another requirement, namely

2. The ability to describe parallel composition of services (i.e. the "|" operator).

The use case demonstrates some of the basic interactions and behaviours that may be required to enable us to describe a choreography between two processes.

3.2.13.10 Notes/Issues

This use case shows a clear problem with potential LiveLock situations. In this example it is possible that any Clienti is locked out from dealing with a server forever.