Session Types Use-cases. (Amazon Collaboration)

Dimitris Kouzapas

1 Introduction

This document lists a number of session types use-cases and examples. Its purpose is to explain to our collaborates in Amazon the syntax and properties of session types.

Before we proceed we need to distinguish between binary session types and multi-party session types. The binary form of session types was the first to be proposed and it describes the interaction between two communicating participants. The multi-party session type theory is a more general theory that was developed to: i) overcome the limitation that binary session types cannot express a well defined communication between a set of participants, and ii) provide a global understanding of multiple participant communication. The multi-party session type theory subsumes binary session types. For the purposes of this document we use the multi-party session types.

1.1 Example Structure

Each example is presented as follows:

1. Sequence diagram. The sequence diagram shows the sequence of message passing interaction between the participants.
2. Global multi-party protocol. We use the syntax of Multi-party protocols to describe the global interaction between participants.
3. Local multi-party protocols. Global multi-party protocols are project to types that describe the local interaction in each of the participants. We use local projection to clarify communication locally.
4. Scribble specification. Scribble [2, 1] is a language developed for the specification of global multi-party session protocols.
5. Session Java implementation. Whenever the use-case scenario is a binary session interaction, we give the code for implementation into Session Java [4, 3]. Session Java is an experimental extension of the Java Language to include static session type checking and runtime session typed communication.

1.2 Examples/Use-cases

We present the following use-cases:

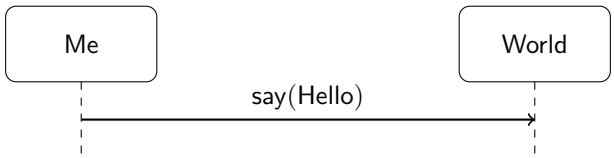
1. Hello World.
2. Travel Agency.
3. On-line Book-store.
4. Post Office Protocol 3.
5. Travel Agency Revisited.

2 Hello World

We introduce the Hello World example for session types. The Hello World is a binary session type where participant Me sends a Hello to the World.

2.1 Sequence Diagram

We give with the very simple to understand sequence diagram for the Hello World example:



2.2 Global Multi-party Protocol

The simplest global multi-party protocol is defined below:

$$G = \mathsf{Me} \rightarrow \mathsf{World} : \{\mathsf{say}(\mathsf{Hello})\}.\mathtt{end}$$

Global multi-party protocols are structured on message passing interactions among participants. The general interaction pattern requires that a participant sends to another participant a labelled annotated message of a specific type.

In the Hello World scenario, participant Me send a message of type Hello annotated with the label say to participant World. The dual way of expressing the same protocol is to see participant World to receive a message Hello on label say from participant Me. After the first interaction takes place, the protocol continues with type end that signifies the end of the communication.

2.3 Local Multi-party Protocols

A global protocol is then *projected* into local session protocols, able to describe the communication taking place in each of the participants. We use the notation $G\lceil \mathtt{p}$ to denote the projection of global protocol G into participant `p`.

$$\begin{aligned} G\lceil \mathsf{Me} &= [\mathsf{World}]!\{\mathsf{say}(\mathsf{Hello})\}; \mathtt{end} \\ G\lceil \mathsf{World} &= [\mathsf{Me}]?\{\mathsf{say}(\mathsf{Hello})\}; \mathtt{end} \end{aligned}$$

Processes and programs that are implemented on local protocols that derived from the projection of global typed, are guaranteed to have *deadlock free* and type matching communication. A third useful property is the *linear usage* of communication endpoints. The term linear usage is used to restrict a resource to be used by a finite set of programs. In the case of session types we require that a communication endpoint is used only by one process at each moment.

2.4 Scribble Specification:

We define the Scribble code for expressing the Hello World protocol:

```
1.  protocol HelloWorld (role Me, role World){
2.      say(Hello) from Me to World;
3.  }
```

Scribble uses a programming language interface to define protocols. A protocol is declared with a number of roles (i.e participants). The basic operation is the `label(MsgType) from RoleA to RoleB`; which is described as: message `MsgType` annotated with label `label` is sent from role `RoleA` role to `RoleB`.

2.5 Binary Session Types

It is usefull to understand the syntax of binary session types since some of the existing tools like Session Java use the syntax of binary session types. The syntax of binary session types is similar to local multiparty session types, with the difference that it ommits the label on the send/receive prefixes and does not include the interaction party on prefixes (i.e the interaction party is implied since the communication is binary).

$$\begin{aligned} \mathsf{Me} &= !\langle \mathsf{String} \rangle; \mathbf{0} \\ \mathsf{World} &= ?(\mathsf{String}); \mathbf{0} \end{aligned}$$

2.6 Session Java

We use Session Java framework to implement the Hello World use-case. The runtime environment of Session Java implements the Socket API, while the compiler performs static checking on communication primitives.

In the Hello World scenario we choose Me as a Client and World as the server (we also omit the `try/catch` clauses).

```
1.  protocol World{begin.?(String)}
2.
3.  void serverSide(int port){
4.      SJServerSocket ss = SJServerSocketImpl.create(World, port);
5.      SJSocket s = ss.accept();
6.      String str = s.recv();
7.  }
```

```
1.  protocol Me{begin.! < String >}
2.
3.  void sayHello(String host, int port){
4.      SJServerAddress sa = SJServerAddress(Me, host, port);
5.      SJSocket s = sa.request();
5.      s.send("Hello");
6.  }
```

At both programs we define the local protocol to be implemented by a Session Java Socket (`SJSocket`). Both local protocols are described in binary session type syntax, in the first line of the corresponding code. Both protocols start with the session initiation keyword `begin`. We use the separator . to separate between session prefixes. Operator !< `String` > defines the send of a String object and it dual operator ?(`String`) defines the receive of a String object.

Class `SJServerSocket` is responsible for creating a server socket that listens to a port and implements the server side session type. Respectively class `SJServerAddress` binds a socket to a session type protocol, a host and a port. Session initiation is done on the pair `ss.accept()`; /`sa.request()`; where session protocol duality is checked at runtime. After session initiation the `s.send("Hello")`; /`s.recv()` duality takes place. Note that the compiler statically checks that the `s.send("Hello")`; /`s.recv()` sequence in a code respects the session type protocol at socket `SJSocket s`.

3 Travel Agency

The Travel Agency use-case is a binary session type protocol of a procedure for booking a trip via a travel agency. In this example we want to clarify choice using labels and recursion on protocols.

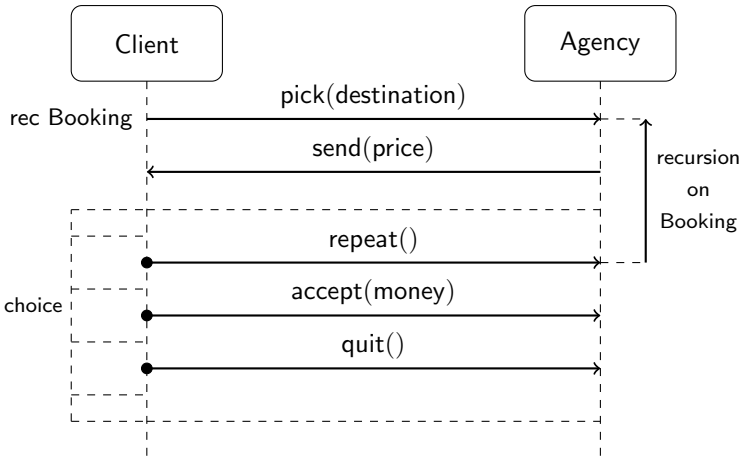
The scenario for the travel agency requires that a Client asks for the price for a destination from the Agency. After receiving the price the client has a set of choices: i) accept the offer, pay and terminate the session; ii) repeat by choosing a new destination; or iii) terminate the session.

3.1 Sequence Diagram

We can see the Client/Agency interaction in the sequence diagram below.

In the sequence diagram of the Travel Agency scenario we use the dotted lines to describe the choices that the Client has. Inside the dotted lines the Client can chose to proceed with one of the proposed actions, which are marked with a black circle at the starting tip of the arrow. The scope of each of the actions can be seen on the *choice* dotted box to the left of the diagram.

The vertical arrow on the right of the diagram, described as *recursion on booking*, is used to denote recursive behaviour.



3.2 Global Multi-party Protocol

We define the multi-party protocol for the Travel Agency:

$$
\begin{aligned}
G \;=\; & \mathsf{rec\ Booking}. \\
& \mathsf{Client} \rightarrow \mathsf{Agency} : \{\mathsf{pick(destination)}\}. \\
& \mathsf{Agency} \rightarrow \mathsf{Client} : \{\mathsf{send(price)}\}. \\
& \mathsf{Client} \rightarrow \mathsf{Agency} : \{\mathsf{accept(money)} : \mathtt{end}, \\
& \qquad\qquad\qquad\qquad \mathsf{repeat}() : \mathsf{Booking}, \\
& \qquad\qquad\qquad\qquad \mathsf{quit}() : \mathtt{end} \\
& \}
\end{aligned}
$$

In this example we introduce the choice and the recursion operators. We use the keyword rec to define recursion label Booking. The Client picks a destination by communicating with the Agency. The Agency then replies with a price. The Client then has the option to choose either to accept(money) the offer and sends the money to the Agency, or repeat() the procedure to pick another destination using the recursive label Booking, or quit() the procedure and finish the protocol on type end.

3.3 Scribble

We use the travel Agency scenario to introduce the recursion and choice operations in Scribble:

```
1.  protocol TravelAgency(role Client, role Agency){
2.      rec Booking{
3.          pick(destination) from Client to Agency;
4.          send(price) from Agency to Client;
5.          choice at Client{
6.              accept(money) :
7.          }
8.          or{
9.              repeat() : Booking
10.         }
11.         or{
12.             quit() :
13.         }
14. }
```

The Scribble language handles recursion using the keyword `rec` to define a recursive label. The scope of the recursive label is then defined inside the following code block, that is annotated using curly brackets {`protocol`}.

The choice construct has the following form:

```
choice at RoleA
       {labelAMsgTypeA : protocol}
    or {labelBMsgTypeB : protocol}
    ⋮
```

The choice construct has an implicit recognition of the receiving role. Also there is a well-formness check so that the choice block have a consistent communication pattern.

In the above scenario the case `accept(money)` sends together with the acceptance the payment for the trip. A more refined expression would be first to accept on label `accept()` and then send the money with the interaction `transaction(money) from Client to Agency`. The second label `repeat()` repeats the procedure for a new choice of a trip using the recursive label `Booking`. The last label `quit()` offers no other interactions and the communication ends.

3.4 Binary Session Types

Before we proceed with Session Java code we give the binary session types for the Travel Agency scenario. Although labels are ommited in the send/receive

prefixes labels are still used for the operations of selection and branching.

$$\begin{array}{rcl} \mathsf{Client} & = & \mu\mathsf{Booking}.!\langle\mathsf{destination}\rangle;?(\mathsf{price}); \\ & & \oplus\{\mathsf{accept} :!\langle\mathsf{money}\rangle;\mathtt{end},\mathsf{repeat} : \mathsf{Booking},\mathsf{quit} : \mathtt{end}\} \\ \mathsf{Agency} & = & \mu\mathsf{Booking}.?(\mathsf{destination});!\langle\mathsf{price}\rangle; \\ & & \&\{\mathsf{accept} :?(\mathsf{money});\mathtt{end},\mathsf{repeat} : \mathsf{Booking},\mathsf{quit} : \mathtt{end}\} \end{array}$$

The notation μBooking. is called primitive recursor and defines recursion. The select $\oplus\{\ldots\}$ and branch $\oplus\{\ldots\}$ are dual and use labels to define the process of one endpoint selecting and the other branching on the selection.

3.5 Session Java

Session Java support binary choice on labels and recursion. We use the Travel Agency scenario to demonstrate this features.

We naturally define Agency as the server and Client as the client.

```
1.    protocol Agency{
2.        begin.
3.        ?[?(String).! < Double >] ∗ .
4.        ?{accept :?(Double), reject :}
5.    }
6.
7.    void bookTrip(){
8.        SJSocket s = connectClient();
8.        s.inwhile(){
9.            String destination = s.recv();
10.            s.send(getprice(destination));
11.        }
12.        s.inbranch(){
13.            case accept : {Double money = s.recv(); }
14.            case reject : {}
15.        }
16.    }
```

The above code is the implementation of the server side in the Travel Agency Scenario. We assume that socket creation and accept happes in method `connectClient`(). The protocol definition uses the structure ?{`label1` : `session1`, `label2` : `session2` $\cdots$} for the receiving part of choice/branching the structure. For the receiving endpoint of recursion we use the structure ?[`session`]$*$.

Session Java uses the keywords `inwhile` and `inbranch` for receiving choices and branching.

The client code follows the dual implementation.

```
1.    protocol Client{
2.        begin.
3.        ![! < String > .?(Double)] ∗ .
4.        !{accept :! < Double >, reject :}
5.    }
6.
7.    void bookTrip(){
8.        SJSocket s = connectServer();
8.        s.outwhile(decideTrip()){
9.            s.send(getDestination());
10.            Double cost = s.recv();
11.       }
12.       if(decided()){
12.           s.outbranch(accept){s.send(money(cost));}
13.       }else{
14.           s.outbranch(reject){}
15.       }
16.   }
```

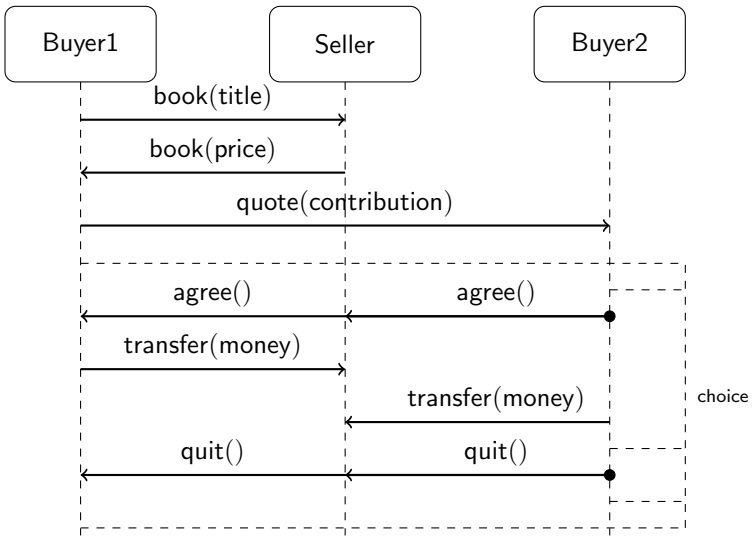
We use dual symbols to define the `Client protocol`. The sending parts of the choice and recursion are now denoted as `outbranch` and `outwhile`.

4 On-line Bookstore

The scenario of the On-line Book store is the first (simple) multi-party protocol. In this example a book buyer asks for the price of a book from an on-line bookstore and then requests money contribution from another buyer to buy the book from the on-line bookstore.

4.1 Sequence Diagram

In this sequence diagram we can clearly see the interactions described after a choice as marked by the dotted *choice* box in the right of the diagram. The point of the choice is marked by a small black circle. Notice on the arrows of the choices that the choices affects more than one participant.



4.2 Global Multi-party Protocol

$$
\begin{aligned}
G \;=\; & \mathsf{Buyer1} \rightarrow \mathsf{Seller} : \{\mathsf{book(title)}\}. \\
& \mathsf{Seller} \rightarrow \mathsf{Buyer1} : \{\mathsf{book(price)}\}. \\
& \mathsf{Buyer1} \rightarrow \mathsf{Buyer2} : \{\mathsf{quote(contribution)}\}. \\
& \mathsf{Buyer2} \rightarrow \mathsf{Buyer1}, \mathsf{Seller} : \{ \\
& \quad \mathsf{agree()} : \mathsf{Buyer1} \rightarrow \mathsf{Seller} : \{\mathsf{transfer(money)}\}. \\
& \qquad\qquad \mathsf{Buyer2} \rightarrow \mathsf{Seller} : \{\mathsf{transfer(money)}\}.\mathtt{end}, \\
& \quad \mathsf{quit()} : \mathtt{end} \\
& \}
\end{aligned}
$$

In this scenario we introduce the possibility of role making a single choice that affects more than one participant. In this cases Buyer2 drives the behaviour of Buyer1 and Seller on to choose either to agree() on the contribution or quit() the session.

4.3 Local Projection

The above multi-party communication protocol is proven to be correct when it is implemented locally on each role. We can clarify the local implementation by giving the local projections.

$$
\begin{aligned}
G\lceil \mathsf{Buyer1} \;=\; & [\mathsf{Seller}]!\{\mathsf{book(title)}\}; \\
& [\mathsf{Seller}]?\{\mathsf{book(price)}\}; \\
& [\mathsf{Buyer2}]!\{\mathsf{quote(contribution)}\}; \\
& [\mathsf{Buyer2}]?\{\mathsf{agree()} : [\mathsf{Seller}]!\{\mathsf{transfer(money)}\}; \mathtt{end}, \\
& \qquad\qquad \mathsf{quit()} : \mathtt{end}\}
\end{aligned}
$$

$$
\begin{aligned}
G\lceil \mathsf{Seller} \;=\; & [\mathsf{Buyer1}]?\{\mathsf{book(title)}\};\\
& [\mathsf{Buyer1}]!\{\mathsf{book(price)}\};\\
& [\mathsf{Buyer2}]?\{\mathsf{agree}() : [\mathsf{Buyer1}]?\{\mathsf{transfer(money)}\};\\
& \qquad\qquad [\mathsf{Buyer2}]?\{\mathsf{transfer(money)}\}; \mathtt{end},\\
& \quad \mathsf{quit}() : \mathtt{end}\}\\
G\lceil \mathsf{Buyer2} \;=\; & [\mathsf{Buyer1}]?\{\mathsf{quote(contribution)}\};\\
& [\mathsf{Buyer1}, \mathsf{Seller}]!\{\mathsf{agree}() : [\mathsf{Seller}]!\{\mathsf{transfer(money)}\}; \mathtt{end},\\
& \qquad\qquad \mathsf{quit}() : \mathtt{end}\}
\end{aligned}
$$

4.4 Scribble

In the next Scribble code segment we can see how a choice that is made locally on a role can affect more than one target roles.

```
1.  protocol Bookstore(role Buyer1, role Buyer2, role Seller){
2.      book(title) from Buyer1 to Seller;
3.      book(price) from Seller to Buyer1;
4.      quote(contribution) from Buyer1 to Buyer2;
5.      choice at Buyer2{
6.          agree() :
7.              transfer(money) from Buyer1 to Seller;
8.              transfer(money) from Buyer2 to Seller;
9.      }
10.     or{
11.         quit() :
12.     }
```

A choice is defined to happen locally on a single role. The targets of the choice are identified implicitly by the Scribble compiler, that also checks for the well-formness each choice and consequently the well-formness of the choice construct.

5 Post Office Protocol 3

More advance scenarios for session types is the description of application layer protocols. In this use-case we uses global multi-party session types to specify a fragment of the Post Office Protocol 3. The POP3 is used by clients to retrieve email data from remote servers. It is a binary server/client protocol.

In the POP3 the Server asks from the client its login data for authentication. In the authentication process an error can happen and the process can start again. If the authentication is successful the protocol proceeds to Transaction where the client can recurse to retrieve its data or it can chose to quit the interaction.

5.1 Sequence Diagram

We present the POP3 as a sequence found in Figure 1 and as a state machine found in Figure 2. POP3 is a more complex protocols that offers a series of nested choices between the Client/Server counterparts.

5.2 Global Multi-party Protocol

We describe POP3 using the global type syntax:

$$
\begin{array}{rcl}
\mathsf{Start} & = & \mathsf{Server} \rightarrow \mathsf{Client} : \{\mathsf{ok}(\mathsf{String})\}.\mathsf{Authorisation} \\
\\
\mathsf{Authorisation} & = & \mathsf{Client} \rightarrow \mathsf{Server} : \{ \\
 & & \quad \mathsf{quit}() : \mathsf{Server} \rightarrow \mathsf{Client} : \{\mathsf{ok}(\mathsf{String})\}.\mathtt{end}, \\
 & & \quad \mathsf{user}(\mathsf{String}) : \\
 & & \qquad \mathsf{Server} \rightarrow \mathsf{Client} : \{ \\
 & & \qquad\quad \mathsf{error}(\mathsf{String}) : \mathsf{Authorisation} \\
 & & \qquad\quad \mathsf{ok}(\mathsf{String}) : \mathsf{Client} \rightarrow \mathsf{Server} : \{ \\
 & & \qquad\qquad \mathsf{quit}() : \mathsf{Server} \rightarrow \mathsf{Client} : \{\mathsf{ok}(\mathsf{String})\}.\mathtt{end} \\
 & & \qquad\quad \} \\
 & & \qquad\quad \mathsf{pass}() : \mathsf{Server} \rightarrow \mathsf{Client} : \{ \\
 & & \qquad\qquad \mathsf{error}(\mathsf{String}) : \mathsf{Authorisation}, \\
 & & \qquad\qquad \mathsf{ok}(\mathsf{String}) : \mathsf{Transaction} \\
 & & \qquad\quad \} \\
 & & \qquad \} \\
 & & \} \\
\\
\mathsf{Transaction} & = & \mathsf{Client} \rightarrow \mathsf{Server} : \{ \\
 & & \quad \mathsf{stat}() : \\
 & & \qquad \mathsf{Server} \rightarrow \mathsf{Client} : \{\mathsf{send}(\mathsf{Int} \times \mathsf{Int})\}.\mathsf{Transaction}, \\
 & & \quad \mathsf{retr}(\mathsf{Int}) : \\
 & & \qquad \mathsf{Server} \rightarrow \mathsf{Client} : \{ \\
 & & \qquad\quad \mathsf{ok}(\mathsf{String}, \mathsf{String}) : \mathsf{Transaction}, \\
 & & \qquad\quad \mathsf{error}(\mathsf{String}) : \mathsf{Transaction} \\
 & & \qquad \} \\
 & & \quad \mathsf{quit}() : \mathsf{Server} \rightarrow \mathsf{Client} : \{\mathsf{ok}(\mathsf{String}) : \mathtt{end}\} \\
 & & \}
\end{array}
$$

6 Travel Agency Revisited

A last example demonstrates the power that session types have on expressing session endpoint delegation. We use the travel agency example to show how a client uses the travel agency as a delegate to book air-plane tickets from the air-plane company. Specifically after the client selects a ticket from a list of offers, the agency delegates its communication endpoint to to the air-plane company. The client continues to interact on the same session but now it communicates directly with the air-plane company to pay the selected ticket.

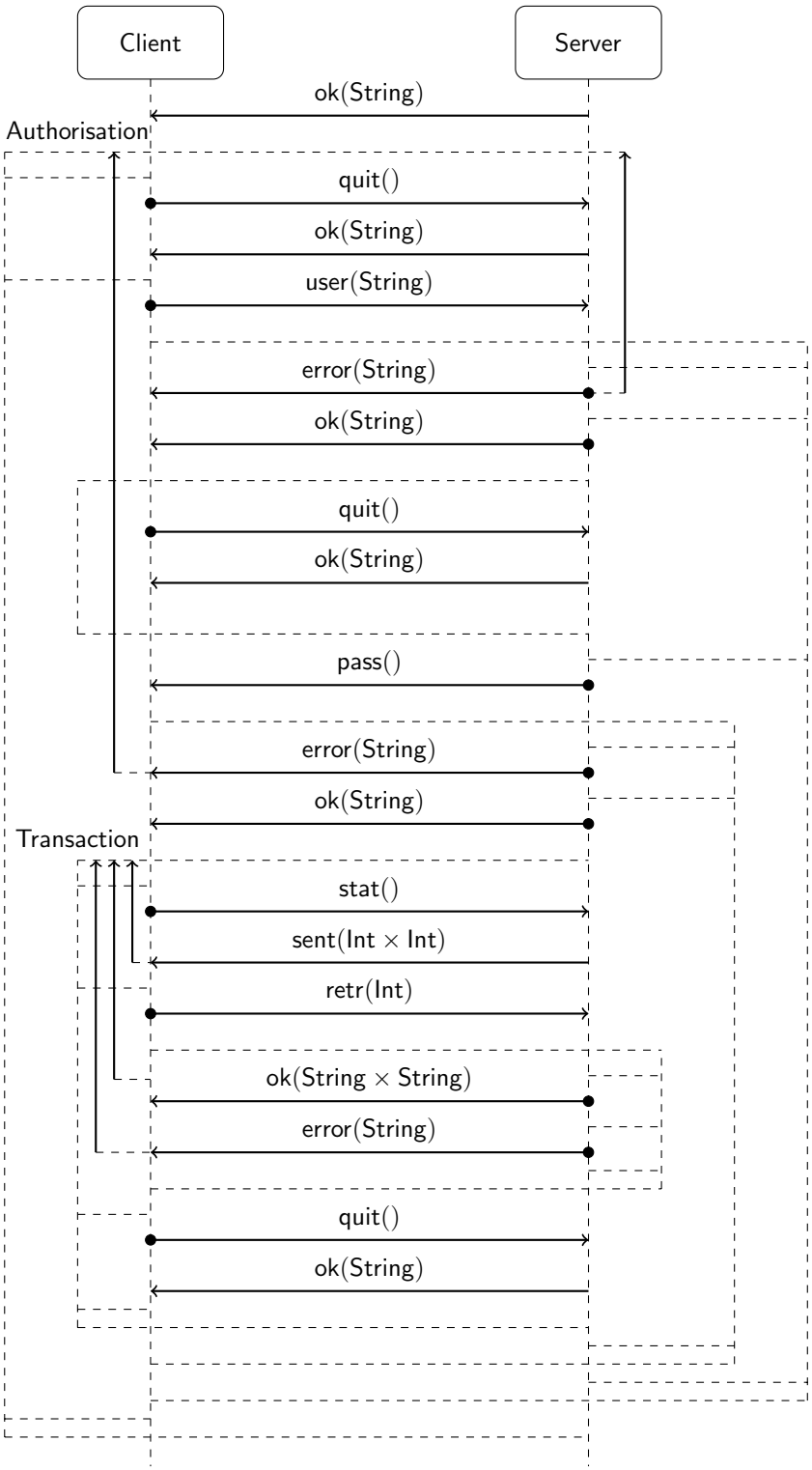


Figure 1: Sequence Diagram: The Post Office Protocol 3

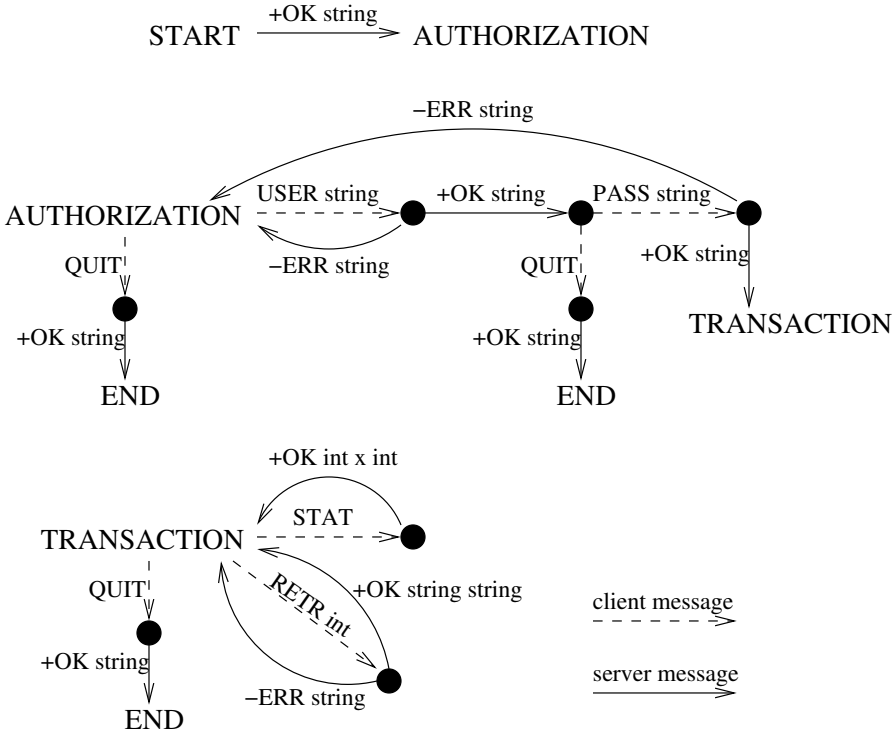


Figure 2: State Machine: The Post Office Protocol 3

6.1 Global Multi-party Protocol

To demonstrate the above scenario we use two binary session protocols. The first protocol G describes the complete interaction between the client and the services he acquires. The second protocol D describes the interaction of delegating the agent's local endpoint to the air-plane company. The local endpoint is a local multi-party protocol.

$$\begin{array}{rcl} G & = & \mathsf{Agent} \to \mathsf{Client} : \{\mathsf{list}(\mathsf{Offers})\}.\mathsf{Client} \to \mathsf{Agent} : \{\mathsf{select}(\mathsf{ticket})\}. \\ & & \mathsf{Agent} \to \mathsf{Client} : \{\mathsf{bank}(\mathsf{account})\}.\mathsf{Client} \to \mathsf{Agent} : \{\mathsf{transfer}(\mathsf{money})\}.\mathtt{end} \\ D & = & \mathsf{Agent} \to \mathsf{Company} : \{\mathsf{delegate}(T)\}.\mathtt{end} \end{array}$$

where

$$T = [\mathsf{Client}]!\{\mathtt{bank}(\mathtt{account})\}; [\mathsf{Client}]?\{\mathtt{transfer}(\mathtt{money})\}; \mathtt{end}$$

Note that type T is a postfix of the local type $G\lceil\mathsf{Agent}$ (local projection of global protocol G into participant Agent).

References

[1] Scribble. a protocol language. `www.scribble.org`.

[2] Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Proceedings of the 7th international conference on Distributed computing and internet technology*, ICDCIT'11, pages 55–75, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *ECOOP*, pages 329–353, 2010.

[4] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP*, pages 516–541, 2008.