Ensuring Faultless Communication Behaviour in an E-Commerce Cloud Application

Rustem A. Kamun and Ross Horne

Kazakh-British Technical University,
Faculty of Information Technologies,
Almaty, Kazakhstan
r.kamun@gmail.com

Abstract

An increasing scope and complexity of Web services raises a new challenge of controlling their interaction. The goal of this work is to ensure that processes in a production Cloud are correctly interacting according to a specification of their communication behaviour. To accomplish this goal, we employ session types to analyse the global and local communication patterns. Session types represents "formal blueprints" of how communicating participants should behave and offers a concise view of the message flows.

This work confirms the feasibility of application of session types on "non-linear" business protocols used by an e-commerce Cloud provider and developed in Session-Java, an extension of Java implementing Session-Based programming. Furthermore, we highlight the importance of this approach for services replicated across multiple Cloud providers each of which must correctly cooperate.

Contents

1 Introduction

The growing needs for information availability and accessibility present new challenges for application development. There are two forces working in parallel with regard to the need

Figure 1: EasyChair logo

for integration. First, the necessity of application integration within a company (enterprise), and second, business-to-business integration. There is an increasingly widespread acceptance of Service-Oriented Architectures (SOA) as a paradigm for integrating software applications within and across organizational boundaries. In this paradigm, independently developed and operated applications are exposed as (Web) services which are then interconnected using a stack of standards, which depicted in Figure 2.

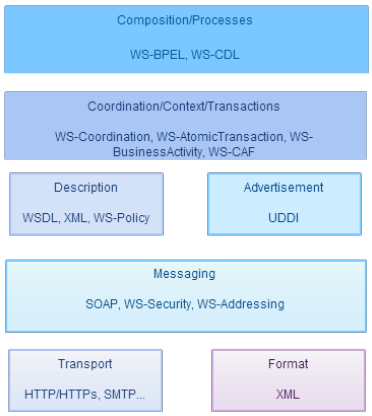

Figure 2: Stack of WS Standards

However there remain open challenges when it comes to managing service interactions that go beyond simple sequences of requests and responses or involve large numbers of participants (multi-party communication). A need arises for new transaction implementations, more suitable for the web. One of those techniques for describing collaboration between a collection of services is a choreography model. It captures the interactions in which the participating services engage and interconnections between these interactions, including control-/data-flow dependencies. However, a choreography does not describe any internal action that occurs within a participating service. In turn, we do not have a clear idea how a global description can be executed, and, therefore, its functionality is not clear.

Massive literature exist on the specification of systems that describe them from the local viewpoint like [4] or [2]. In addition there is no special concept of the participants of communication. It is especially essential in complex interactions that take place today:

1. business transactions with short life span, operating in closely coupled context, where SOA standards may become a trap (online stock exchange(ForEX), services based on BSH protocol[1] such as e-commerce services);
2. applications with a long life span that may result in deadlock.

These are the motivation under the design of WS-CDL (Web Services Choreography Description Language). However, WS-CDL working group discovers its issues that are critically exposed in [1]:

1. tools to validate conformance to choreography specifications to ensure correct cooperation between web services;
2. design time validation and verification of choreographies to guarantee correctness of such properties like deadlock, livelock e.g. behaviour of participants conforms to the choreography interface.

[1]Buyer-Seller-Shipper protocol

Aforementioned challenges can be tackled under a strong theoretical foundations. Fortunately, successful attempts has already made by F. Montesi, M. Carbone who developed Chor programming language [3] based on sessions and theory of trace sets [10], and N. Yoshida, K. Honda developed Session-Java language [6], based on session types [5].

In this paper we will demonstrate a method of controlling process interactions represented by sessions. The formal theory based on session types ensures communication safety by verifying that session implementations of each engaged participant conform to the defined protocol specification. In order to prove the feasibility of this theory we utilized Session-Java, an extension to Java language. It works by specifying the intended process transaction protocol using session types and implementing the interaction using session operations.

In Section 2 we provide fundamentals of SJ syntax and features. Next, in Section 3 we provide detailed explanation of business protocols used by an e-commerce Cloud provider [7]. Finally, in Section **??** we highlight an idea to use session types in designing Intercloud [9] communication protocol.

Through these we are aiming to confirm the suitability of Session-Java as an implementation of business to business transactions. We want to explore the agility and robustness of the language and the scalability as scenarios vary in size but also complexity. In addition we will be looking for things such as ease of programming in SJ, any limitations, bugs or non-implementable scenarios.

2 Basics of Session-Java

Session programming begins from the protocols specification for interaction (using session types), which can then be concretely implemented using a set of structured communication operations available on session sockets. Session programming is applicable for applications where the parties or components cooperate according to specified protocols: session types are formal specifications of such protocols. Session types describe structured sequences of interaction including basic message passing, branching, branching and repetition. A session is an instance of a session type, i.e. the unit of interaction encapsulating one run of a protocol. From the perspective of abstraction, each session, is conducted on a separate channel.

Session programming in SJ consists of the following ordered actions:

1. design specification (protocol) of target communication;
2. mapping protocols into the programs for each participant. For instance, in BSH protocol, we can distinguish three main participants whose actions (processes) are mapped to corresponding programs (software component);
3. By utilizing session programming constructs, implementing the protocol, where each operation is performed as method call;
4. verification of sessions fulfilment by compiler;
5. execution and system testing.

2.1 Protocol Specification

Session programming begins by declaring the protocol for the intended cooperation as follows:

$$\text{protocol name } \{\dots\}$$

where name identifies the protocol, following the standard Java naming rules. The body of the protocol is *session type*, given by the syntax rules in Table 1, below.

Table 1: SJ protocol specification

T	$::= T.T$	Sequencing
	begin	Session initiation
	$! < M >$	Message send
	$?(M)$	Message receive
	$\oplus\{L_1 : T_1, \ldots, L_n : T_n\}$	Session branching
	$\oplus[T]*$	Session iteration.
	rec $L[T]$	Session recursion scope.
	$\#L$	Recursive jump.
	$@p$	Protocol reference.

The session type shows how a session should be designed in terms of actions that the participant should perform. The key point is that the implementation of a session is governed by associated protocol: the SJ compiler (Polyglot[2]). It can be clearly seen from the Table 1, that SJ has enough constructs to describe diverse range of complex interactions: message passing, conditional and repeated expressions. Its worth noting, that each session type element has its dual element, because there is a requirement that two parties implement compatible protocol such as the specification of one party has dual relation to another party.

2.2 Higher Order Communication

In order to describe richer behaviour, SJ has a feature of subtyping. It means that message types can themselves be session types. It also enhances the agility of the type system by allowing the participants in a session to follow different protocols which are compatible [8]. Such communication can be expressed by the following dual constructs:

$$! <?(\text{int}) > \qquad\qquad ?(?(\text{int}))$$

In short, it says that we are expected to send and receive a session of type ?(int). Higher order communication, as we will convince further, is often referred to a session delegation. Figure 3 shows a basic delegation scenario.

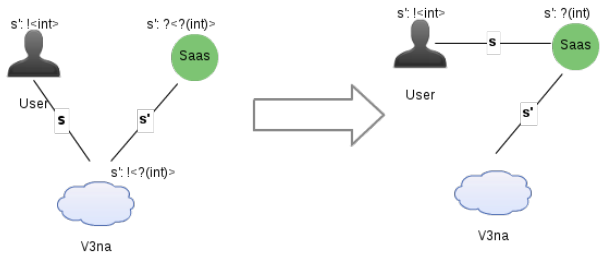

Figure 3: Session delegation

The left part illustrates the session configuration before the delegation is performed: User engaged in a session s of type ! <int> with V3na (Cloud), while Cloud is also involved in a session s' with SaaS of type ! <?(int) >. So, instead of accepting the integer from User himself,

[2]Extensible compile framework available at http://www.cs.cornell.edu/Projects/polyglot/

Cloud delegates his role in s to SaaS, so that he will receive this message. This delegation action corresponds to higher-order send type for the session s' between Cloud and SaaS. The right part of figure illustrates the change in session configuration after the delegation has been performed: User now directly interacting with SaaS for the session s.

2.3 Protocol Implementation using Session

Another meaning for sockets Session sockets are implementing the actual session code according to the specified session type (protocol). They are represent the endpoints (participants) of a session connection: each of the parties owns one endpoint and performs the specified interactions via the SJ session operations on that endpoint. In SJ session sockets are objects that extend the abstract *SJSocket* class. *SJRSocket::SJSocket* and *SJFSocket::SJSocket*, both, employ TCP as underlying transport. SJ is distinguishing session client and server sockets, where the former are used to request sessions from the latter.

Session operations After creating a protocol (session type) and encapsulating the session into SJ socket, it can be implemented within a session-try scope using the session operations depicted in Table 2.

Table 2: Session operations specification

s.request()	begin
s.send(m)	$! < M >$
s.receive()	$?(M)$
s.outbranch(L) {P}	$!\{L : T\}$
s.inbranch() {case L1: P1...case Ln:Pn}	$?\{L_1 : T_1, \ldots, L_n : T_n\}$
s.outwhile(cond) {P}	$[T]*$
s.inwhile() {P}	$?[T]*$
s.recursion(L) {P}	rec $L[T]$
s.recurse(L)	$\#L$

The session operations are invoked via session in a method call-like manner. To delegate a session, the session socket variable must be passed to a send operation on the target session.

```
1 s1.send(s2) // !<T>, where T is the remaining session type of 's2'
```

Only active session sockets can be delegated. The receive operation receives delegated sessions:

```
1 SJSocket s2 = s1.receive()
```

3 Business case studies

V3na.com is an e-commerce web portal that sells SaaS applications for business needs. V3na has developed on Django framework (Python)[3]. The persistence layer is based on MongoDB

[3]Django is a high-level python web framework that encourages rapid development and clean. Further information: https://www.djangoproject.com/

and Memcached. One of the challenging task was to automate the process of SaaS integration. By integration we understand the following processes with a particular SaaS application:

- connection: SaaS user can connect SaaS for trial period by simply clicking on the button;
- subscription extension and freezing;
- payment confirmation;
- one entry point to all user's applications.

Full source code is available at https://github.com/Rustem/Master-thesis.

3.1 Simple Scenario

Protocol declaration As a starting point, let's specify simple business protocol of one of the processes just mentioned, SaaS connection. Informally, it may be interpreted as follows:

1. User begins a request session (s) with cloud service (V3na) and sends the request "Connect SaaS" as JSON-encoded message.
2. V3na sends either:
3. FAIL, if user has no active session (not signed in on V3na) and further interaction terminates
4. OK, if user has logged in and request data has passed validation steps. Then Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage.
5. If OK label take place, Cloud initiates a new session (s') with SaaS and requests it for new user connection with HttpRequestJSONMessage. finally SaaS responds to Cloud with connection status (OK, FAIL) and V3na sends this status to User. Both sessions have to be terminated.

Protocols. The decision in the protocol will be incorporated through the use of outbranch. So the whole scenario is presented on Table 3.

Table 3: Protocols of scenario # 1

User	Cloud (V3na)	SaaS

`protocol p_uv {`
`  begin.`
`  !<JSONMessage>.`
`  ?{`
`  OK: ?(JSONMessage)`
`  .?(int),`
`  FAIL:`
`  }`
`}`
	`p_vu {`
	`  begin.?(JSONMessage)`
	`  .!{`
	`  OK: !<JSONMessage`
	`  >.!<int>,`
	`  FAIL:`
	`  }`
	`protocol http_req_rep {`
	`  !<JSONMessage>.`
	`  ? (JSONMessage)`
	`}`
	`protocol p_vs {`
	`  begin.@http_req_rep`
	`}`
		`protocol p_sv {`
		`  begin.`
		`  ?(JSONMessage). !<`
		`  JSONMessage>`
		`}`

Interactions. The general syntax for global description has been interpreted into a sequence UML diagram, as depicted in Figure 4. The whole syntax is on the down-left side of the figure. In case of choice, terminated branches are out of scope of the main picture, but still a subpart of the whole diagram. Next step is implementation of this diagram in Session-Java.

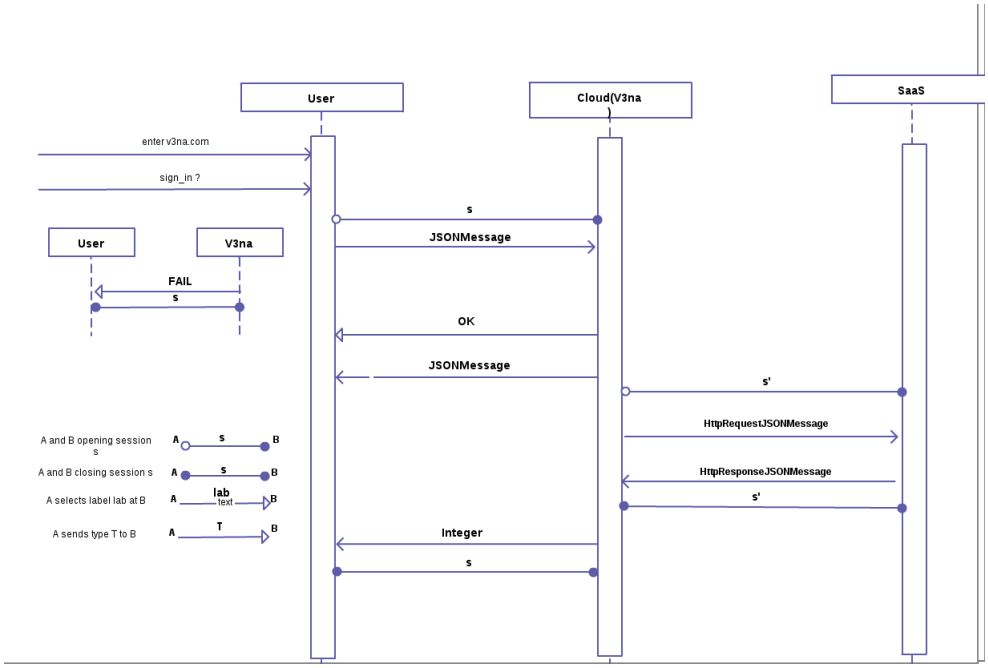

Figure 4: Overview of interactions for Scenario # 1

The main feature of SJ in this scenario is that choice can be expressed by using *outbranch* construct.

3.2 Complicated Scenario

New scenario is a bit harder in complexity. We are going to utilize looping construct as well as demonstrating *session delegation*. The description of the scenario of this subsection presented below as follows:

1. User begins a request session (s) with cloud service (V3na)
2. V3na asks User to login, so next User provides V3na with login and password Strings.
3. V3na receives User credentials and verifies them: If User is authenticated with minimal amount of tries or amount of tries is out of limit, he is allowed to continue further interactions with V3na, otherwise — not. Go back to step 2.
4. If User is not allowed to access V3na, the interaction between User and V3na continues on DENY-branch, otherwise — on ACCESS-branch.
5. If next branch is ACCESS, User sends his connection request with details to V3na. V3na creates new session with SaaS (s') and delegates the remaining session s with User on the latter and sends last user request details. Session s' is terminated.
6. SaaS continues interaction with user by session s. By steps of validation-verification, SaaS either responds User to proceed interaction by branch OK or FAIL. In both cases User

receives from SaaS directly the reason and status of his request. Session s is terminated.

Protocols. First of all, the protocol provided with iterations using $![\ldots]* ?[\ldots]*$. Then protocol introduces higher order operations of type $! < T > \ ? < T >$. Full description is provided in tables 4 and 5.

Table 4: User-Cloud protocols

User	Cloud

`protocol p_uv {`
`  begin.?[!<String>.!<String> ]*.`
`  ?{`
`  ACCESS: !<JSONMessage>.`
`  ?{`
`  OK: ?(JSONMessage), FAIL: ?(JSONMessage)`
`  },`
`  DENY: ?(String)`
`  }`
`}`
	`private protocol p_vu {`
	`  begin.`
	`  ![ ?(String).?(String) // login password`
	`  ]*.`
	`  !{`
	`  ACCESS: ?(JSONMessage).`
	`  !{`
	`  OK: !<JSONMessage>, FAIL: !<JSONMessage>`
	`  },`
	`  DENY: !<String>`
	`  }`
	`}`

Table 5: Cloud-SaaS protocols

Cloud	Saas

`protocol p_vs {`
`  begin.`
`  !< !{`
`  OK: !<JSONMessage>,`
`  FAIL: !<JSONMessage>`
`  } >.!<JSONMessage>`
`}`
	`protocol p_msg {`
	`  !{`
	`  OK: !<JSONMessage>,`
	`  FAIL: !<JSONMessage>`
	`  }`
	`}`
	
	`protocol p_sv {`
	`  begin.?(@p_msg).?(JSONMessage)`
	`}`

Unlike the previous protocol, the Cloud-Saas protocol significantly altered, also authentication process is added to the protocol in interaction between User — Cloud. It is important to note that

```
1 !<!{
2         OK: !<JSONMessage>,
3         FAIL: !<JSONMessage>
4 }>
```

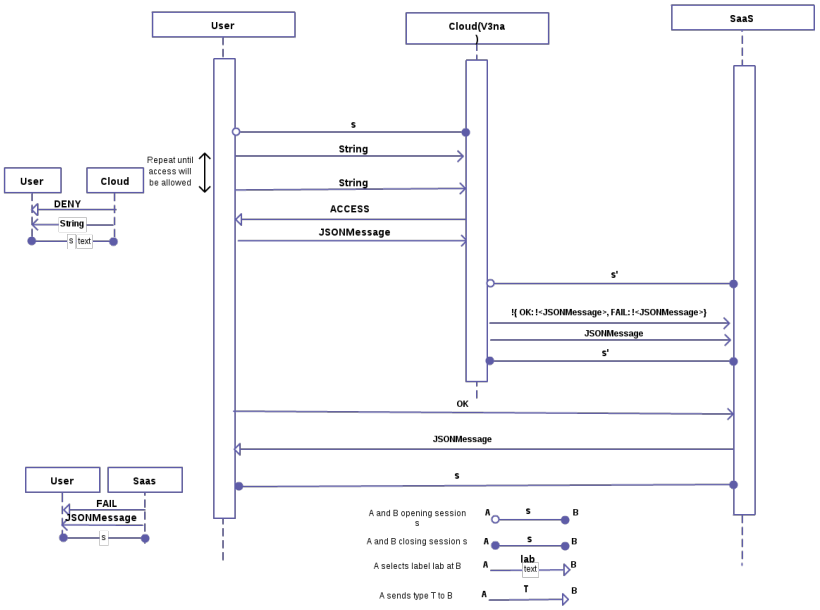

Figure 5: Sequence diagram of interactions for Scenario # 2

corresponds to a higher-order message. The $!< \cdots >$ means that it is the Cloud that is passing the high order message and everything inside it is the protocol of the session that Saas should perform with the User. In Saas — Cloud, the protocol defined in more subtle way containing higher order messages by first defining them and then including them in the protocol. For syntactic convenience, one protocol can be referenced from another using @ operator. The $@p$ is syntactically substituted for the protocol of that name.

Interactions. Figure 3.2 depicts the protocols provided above using an UML sequence diagram. The language of the artifacts has already presented in the first scenario.

Implementation. Despite the fact that session delegation takes place, the program still remains very simple. Actually delegating the protocol is straightforward and only consists of passing the socket to service:

```
1 s_vs.send(user_vu); // pass the remaining protocol
```

I have decided to include the whole segment of code to illustrate the following point.

```
1  user_vu.outbranch(ACCESS) {
2    JSONMessage req_info = user_vu.receive();
3    SJServerAddress addr_vs = SJServerAddress.create(p_vs, saas_hname, saas_port);
4    SJSocket s_vs = SJRSocket.create(addr_vs); try(s_vs) {
5      s_vs.request();
6      s_vs.send(user_vu); // pass the remaining protocol
7      s_vs.send(req_info);
8    } catch(UnknownHostException uhe) {
9      uhe.printStackTrace();
10   }
```

```
11 }
```

To receive a high order message type casting must take place in the case of a protocol, the type of protocol must be explicitly defined:

```
1 v3na_user_socket = (@p_msg) v3na_sv.receive();
```

Where p_msg is defined in the protocols section. This is a reason why it is good practice to first exclusively write the protocol to be delegated and then include it in the final protocol.

References

[1] D. Marlon B. Alistair and O. Phillipa. *A critical overview of the web service choreography description language*. BPTrends, 2005.

[2] J. A. Bergstra and J. W. Klop. *Algebra of communicating processes*. Theoretical Computer Science, 1985.

[3] M. Carbone F. Montesi. Chor - choreography programming language. Available by: http://www.chor-lang.org/.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[5] Nobuko Yoshida Marco Carbone, Kohei Honda and Robin Milner. A theoretical basic of communication-centred concurrent programming. Available at: http://www.eecs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf, 2006.

[6] Kohei Honda Raymond Hu, Nobuko Yoshida. Session-based distributed programming in java. Available at: http://www.doc.ic.ac.uk/~rhu/sessionj/hyh08session-based.pdf, 2008.

[7] Vadim V. Kotov Rustem A. Kamun, German Ilyin. Cloud platform for optimizing your business performance. [online], 2013. Available by http://v3na.com.

[8] Malcolm Hole Simon Gay. Subtyping for session types in the pi calculus. Available at http://goo.gl/Hi5i9, 2005.

[9] Wikipedia. Intercloud. [online], 2013. http://en.wikipedia.org/wiki/Intercloud.

[10] X. Zhao Z. Qiu, C. Cai and H. Yang. *Exploring the essence of choreography*. Proceedings of the 16th international conference on WWW, 2007.