

Ensuring Faultless Communication Behaviour in an E-Commerce Cloud Application

Rustem A. Kamun and Ross Horne

Kazakh-British Technical University, Faculty of Information Technology,
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.

1 Introduction

The need for distributed highly available services presents challenges for application development. It is necessary for applications to be integrated both within an enterprise, and between businesses. Service-Oriented Architectures (SOA) are widely accepted as a paradigm for integrating software applications within and across organizational boundaries. In this paradigm, independently deployed applications are exposed as Web Services which are then interconnected using a stack of standards (depicted in Figure 1).

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 technique for describing collaboration between a collection of services is a choreography model. Choreographies capture the interactions in which the participating services engage and interconnections between these interactions, including control-/data-flow dependencies. However, a choreography does not describe internal effects within a participating service. Furthermore, a choreography does not specify how a global description can be executed.

Much literature exist on the specification of systems that describe services from the local viewpoint [13,5]. The concept of a participant in a communication is essential in complex interactions. Applications include business transactions with short life span, operating in closely coupled context (e.g. the online stock exchange (ForEX), and e-commerce services based on Buyer-Seller-Shipper (BSH) protocols). Although, in

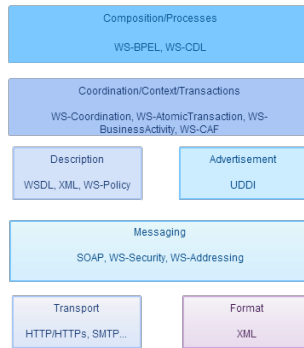


Fig. 1. Stack of WS Standards

closely coupled settings, the SOA standards may incur a significant performance overhead. Highly available services are likely to have a long life span that may result in deadlock.

Such challenges motivated the design of Web Services Choreography Description Language (WS-CDL) [8]. The WS-CDL working group identified critical issues [2] including:

1. the need for 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.

The aforementioned challenges can be tackled by adopting a solid foundational model. Successful approaches based on session types [8,7] include: the Chor and Jolie programming languages of Carbone and Montesi [16,9] based on sessions and trace sets [20]; Session-Java [15], Scribble [14] and Session C [18] due to Honda and Yoshida; Sing# [3] that extend Spec# with choreographies; and UBF(B) [1] for Erlang.

In this paper, we 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 evaluate the feasibility of this theory we use Session-Java, an extension to Java language. Session Java works by specifying the intended process transaction protocol using session types and implementing the interaction using session operations.

In Section 2 we provide an overview of SJ. In Section 3, we provide detailed explanation of business protocols used by an e-commerce Cloud provider. Finally, in Section 4, we highlight how session types can improve the design of Intercloud [6] communication protocols.

2 Basics of Session-Java

Session programming begins from the protocols specification for interaction (using session types), which can then be concretely implemented using 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 and recursion. 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 protocols, 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:

protocol name { ... }

where name identifies the protocol, following the standard Java naming rules. The body of the protocol defines a *session type*, according to the grammar in Figure 2.

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

Fig. 2. SJ protocol specification

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 (Polyglot¹). It can be clearly seen from the Table 2, that SJ has enough constructs to describe a 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 [22]. Such communication can be expressed by the following dual constructs:

$$!\langle ?(\text{int}) \rangle \quad ?(\langle ?(\text{int}) \rangle)$$

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

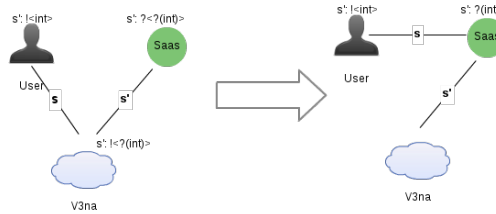


Fig. 3. Session delegation

The left part illustrates the session configuration before the delegation is performed: User engaged in a session s of type $!\langle \text{int} \rangle$ with V3na (Cloud), while Cloud is also involved in a session s' with SaaS of type $!\langle ?(\text{int}) \rangle$. So, instead of accepting the integer from User himself, 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 (par-

¹ <http://www.cs.cornell.edu/Projects/polyglot/>. Extensible compile framework.

ticipants) 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 Figure 4.

<i>s.request()</i>		begin
<i>s.send(m)</i>		!(<i>M</i>)
<i>s.receive()</i>		?(<i>M</i>)
<i>s.outbranch(L) {P}</i>		!{ <i>L:T</i> }
<i>s.inbranch() {caseL1:P1 ... caseLn:Pn}</i>		?{ <i>L₁: T₁, ..., L_n: T_n</i> }
<i>s.outwhile(cond) {P}</i>		[<i>T</i>]*
<i>s.inwhile() {P}</i>		?[<i>T</i>]*
<i>s.recursion(L) {P}</i>		rec <i>L</i> [<i>T</i>]
<i>s.recurse(L)</i>		# <i>L</i>

Fig. 4. SJ protocol specification

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.com² is an e-commerce Web portal that sells SaaS applications for business needs. V3na has developed on Django framework — a high-level Web framework for python. The persistence layer is based on MongoDB 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 application³:

² <http://v3na.com>. Cloud platform for optimizing your business performance.

³ Source code is available at <https://github.com/Rustem/Master-thesis>.

- 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.

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 out-branch. So the whole scenario is presented on Table 1.

Table 1: Protocols of scenario # 1

User	Cloud (V3na)	SaaS
<pre> protocol p_uv { begin. !<JSONMessage>. ?{ OK: ?(JSONMessage).?(int), FAIL: } } </pre>	<pre> p_vu { begin.?(JSONMessage) .!{ OK: !< JSONMessage >.!<int>, FAIL: } protocol http_req_rep { !<JSONMessage>. ? (JSONMessage) } protocol p_vs { begin. @http_req_rep } } </pre>	<pre> protocol p_sv { begin. ?(JSONMessage) . !<JSONMessage > } </pre>

Interactions. The general syntax for global description has been interpreted into a sequence UML diagram, as depicted in Figure 5. 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.

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.

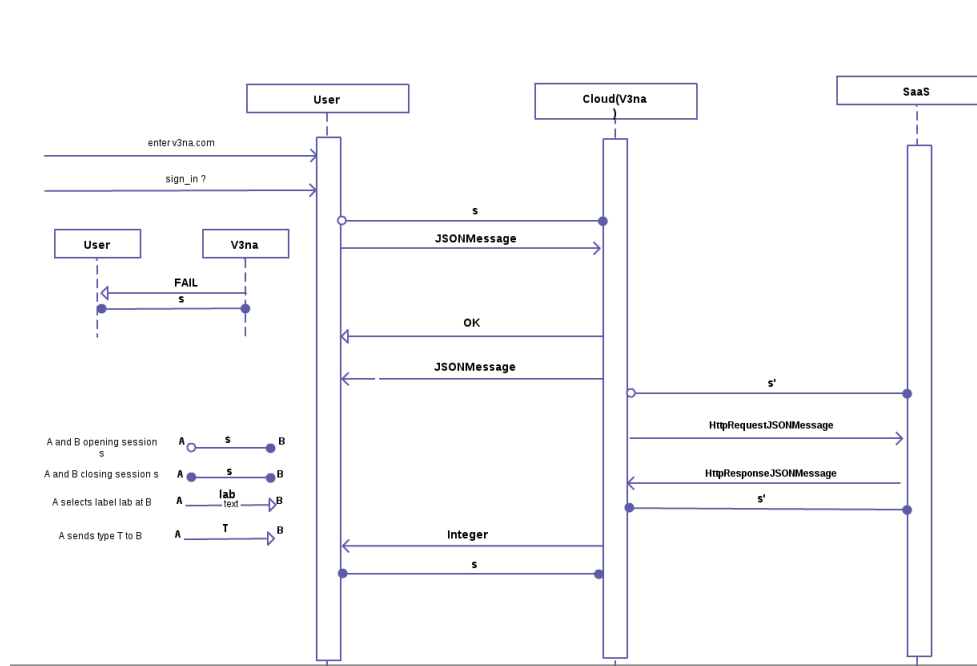


Fig. 5. Overview of interactions for Scenario # 1

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 $![\dots]*?[\dots]*$. Then protocol introduces higher order operations of type $! < T > ? < T >$. Full description is provided in tables 2 and 3.

Table 2: User-Cloud protocols

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

Table 3: Cloud-SaaS protocols

Cloud	SaaS
<pre> protocol p_vs { begin. !< !{ OK: !<JSONMessage>, FAIL: !<JSONMessage> } >.!<JSONMessage> } </pre>	<pre> protocol p_msg { !{ OK: !<JSONMessage>, FAIL: !<JSONMessage> } } protocol p_sv { begin.?(@p.msg).?(JSONMessage) } </pre>

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>

corresponds to a higher-order message. The $!< \dots >$ 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.

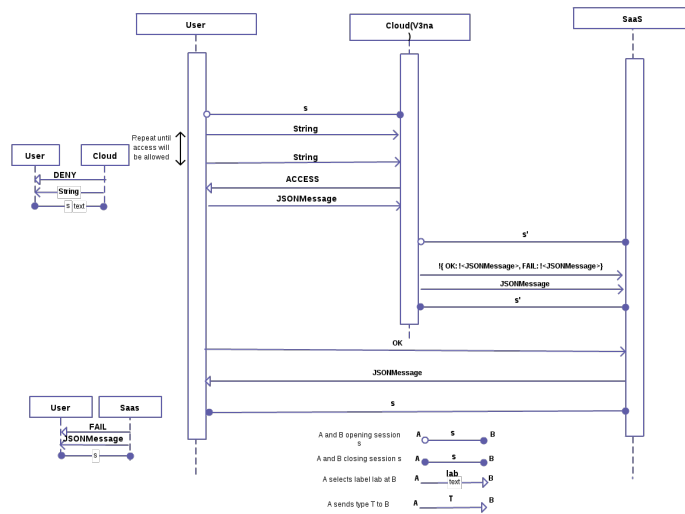


Fig. 6. Sequence diagram of interactions for Scenario # 2

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 = SJSocket.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.

4 Future Work: InterCloud and Session Types

Cloud computing is moving to the concept where cloud operated by one enterprise interoperating with a clouds of another is powerful idea. So far that is limited to use cases where code running on one cloud explicitly references a service on another cloud. There is no implicit and transparent interoperability. Different visions has already proposed in papers [21,11,10]. The most full picture of cloud inter-networking is depicted by [21]. They emphasized the main components of general inter-networking architecture: (a) *Cloud Coordinator*, for bringing out Cloud services; (b) *Cloud Broker*, “for mediating between service consumers and Cloud coordinators”; (c) *Cloud Exchange* (e.g. Cloud Integrator), for collecting consumers’ demands and locating Cloud providers with them with offers.

Our approach is to employ multiparty session types [18] for type-safe conversation between Cloud providers and Cloud Integrators (many-to-many conversation). It starts by specifying the intended interactions as an inter Cloud protocol in the, contract checker, UBF. Then processes for each role (either Cloud provider or Cloud Exchange) are implemented in Erlang or Python (they are best for working with high load applications). Since all the roles should be aware of each other in global network in order to dialog with each other, we are going to use SockJS⁴ protocol for presence and AMQP messaging (it’s thin, flexible). Moreover, there is an idea, to extend SockJS protocol during communication initiation to check at runtime that each interaction is correct and as a result the whole communication is safe. As a starting point for dynamic verification observers, we referred to [17] work.

⁴ SockJS is an effort to define a protocol between in-browser SockJS-client and its server-side counterparts

5 Conclusion

In today's time, there is an actual need to develop structured, high-level and type-safe techniques for programming and interaction. The presented case studies demonstrated how communication safety can be guaranteed through a combination of static and dynamic type verification. Although session-based programming is on its infancy, it's developed with huge steps and already proved its feasibility in various fields: parallel algorithms [19], event-driven programming [4], multiparty conversations [12].

References

1. Joe Armstrong. Getting Erlang to talk to the outside world. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 64–72. ACM, 2002.
2. Alistair Barros, Marlon Dumas, and Phillipa Oaks. A critical overview of the web services choreography description language. *BPTrends Newsletter*, 3:1–24, 2005.
3. Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. *ACM SIGPLAN Notices*, 47(1):191–202, 2012.
4. Andi Bejleri, Raymond Hu, and Nobuko Yoshida. Session-based programming for parallel algorithms. 2010.
5. Jan A. Bergstra and Jan Willem Klop. Algebra of communicating processes with abstraction. *Theoretical computer science*, 37:77–121, 1985.
6. David Bernstein, Erik Ludvigson, Krishna Sankar, Steve Diamond, and Monique Morrow. Blueprint for the intercloud-protocols and formats for cloud computing interoperability. In *ICIW'09.*, pages 328–336. IEEE, 2009.
7. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for Web services. In *Programming Languages and Systems*, pages 2–17. Springer, 2007.
8. Marco Carbone, Kohei Honda, Nobuko Yoshida, Robin Milner, Gary Brown, and Steve Ross-Talbot. A theoretical basis of communication-centred concurrent programming. WS-CDL working report, W3C, 2006.
9. Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: Multiparty asynchronous global programming. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 263–274. ACM, 2013.
10. E. Cavalcante, F. Lopes, T. Batista, N. Cacho, F.C. Delicato, and P.F. Pires. Cloud integrator: Building value-added services on the Cloud. In *Network Cloud Computing and Applications (NCCA'11)*, pages 135–142, 2011.
11. Vij D. Bernstein. Using xmpp as a transport in intercloud protocols. In *In Proceedings of CloudComp 2010, the 2nd International Conference on Cloud Computing*, pages 973–982. CiteSeer.
12. Michael Emmanuel. Session-based web service programming for business protocols. Available at: http://www.doc.ic.ac.uk/~yoshida/CDL_USECASE/sj/report/report.pdf.
13. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
14. Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In *Distributed Computing and Internet Technology*, pages 55–75. Springer, 2011.
15. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in java. In *ECOOP 2008*, pages 516–541. Springer, 2008.

16. Fabrizio Montesi and Marco Carbone. Chor — choreography programming language. <http://www.chor-lang.org/>.
17. Romyana Neykova. Session types go dynamic or how to verify your Python conversations. In *PLACES'13. Rome, Italy, 23 March*, pages 34–39, 2013.
18. Nicholas Ng, Nobuko Yoshida, and Kohei Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *Objects, Models, Components, Patterns*, pages 202–218. Springer, 2012.
19. Nicholas Ng, Nobuko Yoshida, Olivier Pernet, Raymond Hu, and Yiannos Kryptis. Safe parallel programming with Session Java. In *COORDINATION*, pages 110–126, 2011.
20. Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th international conference on World Wide Web*, pages 973–982. ACM, 2007.
21. Rajiv Ranjan Rajkumar Buyya and Rodrigo N. Calheiros. InterCloud: Utility-oriented federation of Cloud Computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.
22. Malcolm Hole Simon Gay. Subtyping for session types in the pi calculus. *Journal Acta Informatica*, 42(2-3):191–225, 2005.