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. The scale and complexity of Web Services raises the 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.

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 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 extends Spec# with choreogrphies; 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

We briefly outline how Session-Java is employed to correctly implement protocols. Firstly, the global protocol is specified using sequence diagrams. The global specification is projected to sessions types, which specify the protocol for each participant. The session is then implemented using operations on session sockets. The correctness of the protocol can be verified using session types.

2.1 Protocol Specification

The body of a protocol defines a *session type*, according to the grammar in Figure 1. The session type specifies the actions that the participant in a session should perform. In SJ, the behaviour of an implementation of a session is monitored by the associated protocol, as enforced by the SJ compiler (Polyglot[1]). The constructs in Figure 1 can describe a diverse range of complex interactions, including message passing, branching

[1] `http://www.cs.cornell.edu/Projects/polyglot/`. Extensible compile framework.

and iteration. Each session type construct has its dual construct, because a typical requirement is that two parties implement compatible protocols such that the specification of one party is dual to another party.

L_1, L_2 tag

p protocol name

$M ::= \mathit{Datatype} \mid T$ message

$S ::= p\,\{T\}$ session

$$
\begin{array}{rll}
T ::= & T\,.\,T & \text{Sequencing} \\
\mid & \texttt{begin} & \text{Session initiation} \\
\mid & !\langle M \rangle & \text{Message send} \\
\mid & ?(M) & \text{Message receive} \\
\mid & \{L_1 : T_1, \ldots, L_n : T_n\} & \text{Session branching} \\
\mid & [T]\,* & \text{Session iteration} \\
\mid & \texttt{rec}\,L\,[T] & \text{Session recursion scope} \\
\mid & \#L & \text{Recursive jump} \\
\mid & @\,p & \text{Protocol reference}
\end{array}
$$

Fig. 1: SJ protocol specification

Higher Order Communication. SJ allows message types to themselves be session types. This is called higher-order communication and is supported by using subtyping [22]. Consider the dual constructs $!\langle ?(\text{int})\rangle$ and $?(?(\text{int}))$. These types specify sessions that expects to respectively send and receive a session of type $?(\text{int})$. Higher order communication is often referred to a session delegation. Figure 2 shows a basic delegation scenario.

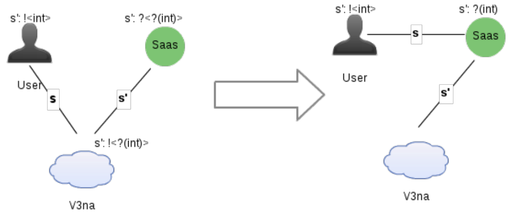

Fig. 2: Session delegation

In Figure 2, the left diagram represents the session configuration before the delegation is performed: the user is engaged in a session s of type $!\langle\text{int}\rangle$ with the Cloud (V3na[2]), while the Cloud is also involved in a session s' with SaaS of type $!\langle ?(\text{int})\rangle$. So, instead of accepting the integer from the user, Cloud delegates his role in s to SaaS. The diagram on the right of Figure 2 represents the session configuration after the delegation has been performed: the user now directly interact with SaaS for the session s. The

[2] http://v3na.com. Cloud platform for optimizing your business performance.

delegation action corresponds to a higher-order send type for the session s' between Cloud and SaaS.

Protocol Implementation using Session-Java. Session sockets represent the participants of a session connection. Each sockets implement the session code according to the specified session type. In SJ session sockets extend the abstract *SJSocket* class. The session is implemented within a session-try scope using specific session operations.

To delegate a session, the session socket variable must be passed to a send operation on the target session. For example, assuming that $s2$ is an active session of type T, then the type of $s1.send(s2)$ is $!\langle T \rangle$. The corresponding receive operation, *SJSocket* $s2 = s1.receive()$ receives delegated sessions.

3 Case Study: Protocols for a Cloud

Our case study is an e-commerce Web portal called V3na that sells SaaS applications for business needs V3na was developed[3] in the Django framework — a high-level Web framework for Python. The persistence layer is based on MongoDB and Memcached. A major challenge was to automate the process of SaaS integration. In particular, V3na implements the following problems that can be addresses using sessions types:

- A SaaS user can connect to SaaS for trial period by simply clicking on the button;
- V3na provides one entry point to all of the user's applications.
- A subscription may be extended or frozen;
- The payment for use of a service can be confirmed confirmation;

3.1 A Simple Scenario with Branching

To begin, we specify a simple business protocol for SaaS connection. The protocol is informally specified 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:
   (a) FAIL, if user has no active session (not signed in on V3na) and further interaction terminates
   (b) 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 message details in JSONMsg.
3. If OK label take place, Cloud initiates a new session (s') with SaaS and requests it for new user connection with message details in JSONMsg.
4. Finally, the SaaS responds to the Cloud with connection status OK or FAIL and V3na sends this status to the user. Both sessions are then terminated.

In Figure 3 we depict the protocol specifications for each participant (Cloud, SaaS or User). The protocols between User and Cloud and Cloud and SaaS are dual, i.e. the specification of interaction from one perspective is opposite from another.

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

Protocol 1: User

```
protocol p_uv {
  begin.
  !<JSONMsg>.
  ?{
    OK: ?(JSONMsg).?(int
         ),
    FAIL:
  }
}
```

Protocol 2: Cloud

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

Protocol 3: SaaS

```
protocol p_sv {
  begin.
  ?(JSONMsg).!<JSONMsg>
}
```

Fig. 3: Protocol specifications for Scenario 1

The global description of the protocol is presented as a sequence diagram in Figure 4. To represent choice, the OK case is represented by the main picture and the FAIL case is a sub-diagram. We implement this diagram in Session-Java, where the branching is implemented employing the `outbranch` and `inbranch` operations.

3.2 The Scenario extended with Looping

We now introduce the looping construct and demonstrate *session delegation*. The informal description of the scenario is 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.

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 figures 5 and 6.

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

```
!<!{
```

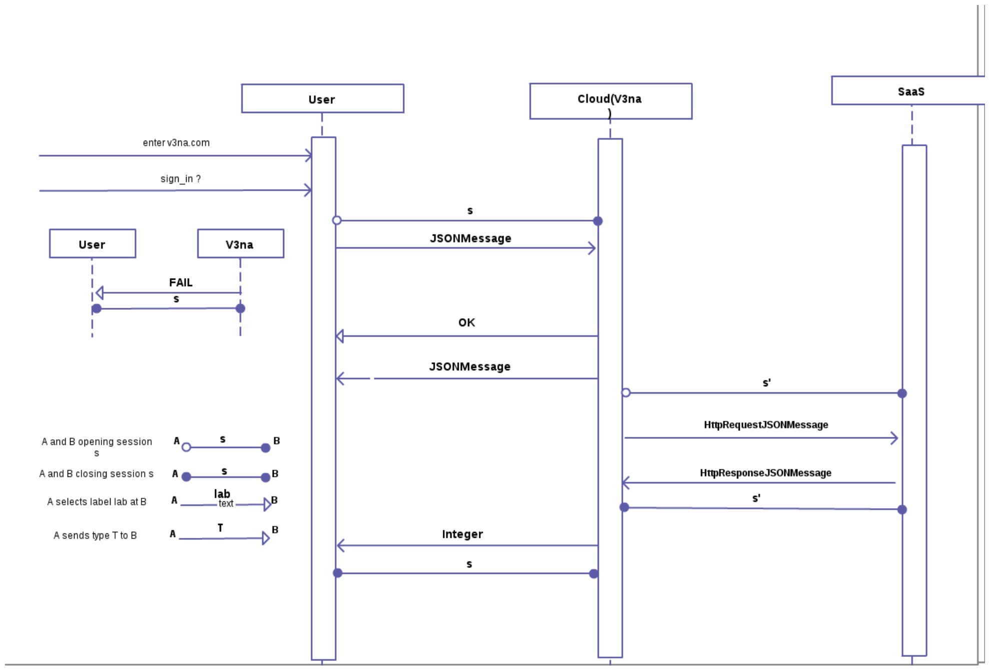

Fig. 4: Overview of interactions for Scenario # 1

```
            OK:  !<JSONMessage>,
             FAIL:  !<JSONMessage>
}>
```

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 **??** 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:

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

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

Protocol 1: User

```
protocol p_uv {
  begin.?[!<String>.!<String> ]*.
  ?{
   ACCESS: !<JSONMsg>.
   ?{
     OK: ?(JSONMsg), FAIL: ?(JSONMsg)
   },
  DENY: ?(String)
  }
}
```

Protocol 2: Cloud

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

Fig. 5: User-Cloud interaction protocol specifications for Scenario 2

Protocol 1: Cloud

```
protocol p_vs {
  begin.
  !< !{
    OK: !<JSONMsg>,
    FAIL: !<JSONMsg>
  } >.!<JSONMsg>
}
```

Protocol 2: Cloud

```
protocol p_msg {
  !{
    OK: !<JSONMsg>,
    FAIL: !<JSONMsg>
  }
}

protocol p_sv {
  begin.?(@p_msg).?(JSONMsg)
}
```

Fig. 6: Cloud-SaaS interaction protocol specifications for Scenario 2

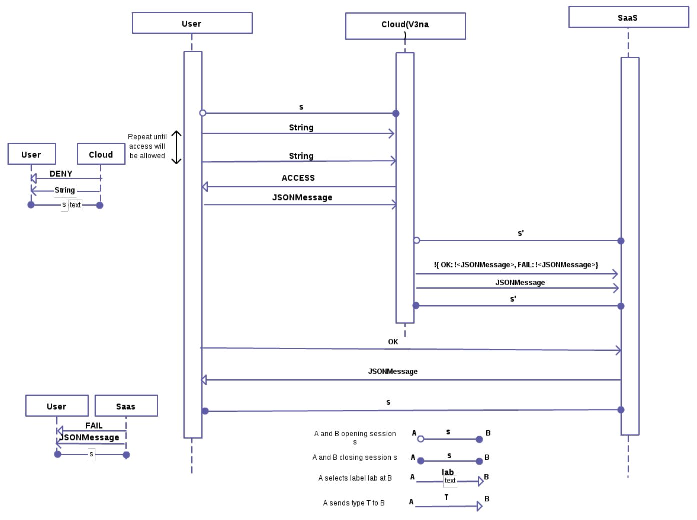

Fig. 7: Sequence diagram of interactions for Scenario # 2

```
user_vu.outbranch(ACCESS) {
  JSONMessage req_info = user_vu.receive();
  SJServerAddress addr_vs = SJServerAddress.create(p_vs,
      saas_hname, saas_port);
  SJSocket s_vs = SJRSocket.create(addr_vs); try(s_vs) {
    s_vs.request();
    s_vs.send(user_vu); // pass the remaining protocol
    s_vs.send(req_info);
  } catch(UnknownHostException uhe) {
    uhe.printStackTrace();
  }
}
```

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:

```
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 Brocker*, "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 SockJS[4] 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

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]. In this paper, we demonstrated the power of session types to control interaction patterns between communicating processes. The static type-checking in compile time and dynamic type-checking in connection initiation time ensures protocols compatibility. Higher level of abstraction of session types, shown in SJ language, prove its feasibility to effortlessly translate medium Business scenarios into protocols. Due to support of high-level communication (session delegation), it have to be remarked how seamless was the refinement process from Scenario 1 to Scenario 2. In addition, we omit more complicated Scenario about payment and wallet recharging transactions (available by `https://github.com/Rustem/Master-thesis`), where we discovered the benefits of combining session delegation and threading provided in SJ. Finally, we hope that our feedback about session types will be as a solid starting point for further research in this area.

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. Rumyana 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 Kryftis. 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.