Ensuring Faultless Communication Behaviour in a Commercial Cloud

Ross Horne[1,2] and Timur Umarov[2]

[1] Romanian Academy, Institute of Computer Science,
Blvd. Carol I, no. 8, 700505 Iaşi, Romania
[2] Faculty of Information Technology, Kazakh-British Technical University,
Tole Bi 59, Almaty, Kazakhstan
ross.horne@gmail.com t.umarov@kbtu.kz

Abstract. For many Cloud providers, the backbone of their system is a Cloud coordinator that exposes a portfolio of services to users. The goal of this work is to ensure that a Cloud coordinator interacts correctly with services and users according to a specification of their communication behaviour. To accomplish this goal, we employ session types to analyse the global and local communication patterns. A session type provides an appropriate level of abstraction for specifying message exchange patterns between participants. This work confirms the feasibility of applying session types to protocols used by a commercial Cloud provider. The protocols are developed in SessionJ, an extension of Java implementing session-based programming. We also highlight that the same techniques can be applied when Java is not the development environment by type checking runtime monitors, as in Scribble. Finally, we suggest how our methodology can be used to ensure the correctness of protocols for Cloud brokers, that integrate services exposed by multiple Cloud coordinators, each of whom must correctly cooperate with the Cloud broker.

Keywords: session types, runtime monitors, Cloud, Intercloud

1 Introduction

Cloud providers typically offer a portfolio of services, where access and billing for all services are integrated in a single distributed system. The integration of services is done by a Cloud coordinator or controller [1,2,3] that exposes services to users. Services are made available on demand to anyone with a credit card, eliminating the up front commitment of users [4]. Furthermore, there is a drive for services to be integrated, not only within a Cloud, but also between multiple Cloud providers.

For a Cloud coordinator that integrates heterogeneous services with a single point of access and billing strategy, protocols can become complex. Thus we require an appropriate level of abstraction to specify and implement such protocols. Further to the complexity, the protocols are a critical component of the business strategy of a Cloud provider. Failure of the protocols could result in divergent behaviour that jeopardises services, potentially leading to loss of customers and legal disputes. These risks can be limited by using techniques that statically prove that protocols are correct and dynamically check that protocols are not violated at runtime.

It is challenging to manage service interactions that go beyond simple sequences of requests and responses and involve large numbers of participants. One technique for managing protocols between multiple services is to specify the protocol using a choreography. A choreography specifies a global view of the interactions between participating services. However, by itself, a choreography does not determine how the global view can be executed.

The challenge of controlling interactions of participants motivated The WS-CDL working group to identify critical issues [5]. One issue is the need for tools to validate conformance of participants to a choreography specification, to ensure that participants cooperate according to the choreography. Another issue is the static design time verification of choreographies to analyse safety properties such as the absence of deadlock or livelock in a system.

The aforementioned challenges can be tackled by adopting a solid foundational model, such as session types [6,7]. Successful approaches related to session types include: SessionJ [8,9], Session C [10] and Scribble [11] due to the team lead by Honda and Yoshida; Sing# [12] that extends Spec# with choreographies; and UBF(B) [13] for Erlang.

In this paper, we present a case study where the interaction of process that integrate services in a commercial Cloud provider[3] are controlled using session types. Session types ensure communication safety by verifying that session implementations of each participant (the customers, services and Cloud coordinator), conform to the specified protocols. In our case study, we use SessionJ, an extension of Java supporting sessions, to specify protocols used by the Cloud coordinator that involve branching, iterative behaviour and higher order communication.

In Section 2 we describe a methodology for designing protocols in SessionJ. In Section 3, we introduce and refine a protocol used by a Cloud coordinator which is implemented using SessionJ. Finally, in Section 4, we suggest that session types can be used in the design of reliable Intercloud protocols, following the techniques employed in this work.

2 Methodology for Verifying Protocols in SessionJ

We chose SessionJ for the core of our application, since Java was already used for several services. SessionJ has a concise syntax that tightly extends Java socket programming. Furthermore, the overhead of runtime monitoring in SessionJ is low [8,9].

We briefly outline a methodology for using SessionJ to correctly implement protocols. Firstly, the global protocol is specified using a global calculus similar to sequence diagrams. Secondly, the global calculus is projected to sessions types, which specify the protocol for each participant. Thirdly, the session is implemented using operations on session sockets. The correctness of the global protocol can be verified by proving that the implementation of each session conforms to the corresponding session type.

Protocol Specification. The body of a protocol is defined as a *session type*, according to the grammar in Figure 1. The session type specifies the actions that the participant

[3] V3na Cloud Platform. AlmaCloud Ltd., Kazakhstan. http://v3na.com

L_1, L_2	label
p	protocol name
$M ::= Datatype \mid T$	message
$S ::= p\,\{T\}$	protocol

$T ::= T\,.\,T$	sequencing
$\mid$ `begin`	session initiation
$\mid\ !\langle M \rangle$	message send
$\mid\ ?(M)$	message receive
$\mid\ !\{L_1 : T_1, \ldots, L_n : T_n\}$	branching send
$\mid\ ?\{L_1 : T_1, \ldots, L_n : T_n\}$	branching receive
$\mid\ ![T]*$	iterative send
$\mid\ ?[T]*$	iterative receive
$\mid\ @p$	protocol reference

Fig. 1: SessionJ protocol specification using session types (T).

in a session should perform. The constructs in Figure 1 can describe a diverse range of complex interactions, including message passing, branching 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.

Higher Order Communication. SessionJ allows message types to themselves be session types. This is called higher-order communication and is supported by using subtyping [14]. Consider the dual constructs $!\langle ?(\text{int}) \rangle$ and $?(?(\text{int}))$. These types specify sessions that expect to respectively send and receive a session of type $?(\text{int})$. Higher order communication is often referred to as session delegation. Figure 2 shows a basic delegation scenario.

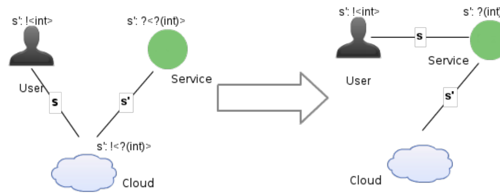

Fig. 2: Session delegation.

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, while the Cloud is also involved in a session s' with a service of type $!\langle ?(\text{int}) \rangle$. So, instead of accepting the integer from the user, the Cloud delegates its role in session s to the service. The diagram on the right of Figure 2 represents the session configuration after the delegation has been performed: the user now directly interacts with the service for the session s. The delegation action corresponds to a higher-order send type for the session s' between the Cloud and the service.

Protocol Implementation and Runtime Monitors. Session sockets represent the participants of a session. Each socket implements the session code according to the specified session type, using a vocabulary of session operations. The session is implemented within a session-try scope, which allows the implementation to respond to exceptions thrown by a runtime monitor.

The runtime monitor dynamically checks the types of messages receieved, since in a distributed system it is difficult to guarantee that other participants always send a message of the type specified. The runtime also detects the failure of any participant to enact its role in the session. Upon failure, a meaningful exception is raised that can be used to elegantly recover or close a failed session. At the scales which Cloud providers operate, unavoidable node failures are expected to frequently occur. For example, during a MapReduce job over a cluster of 130 nodes, it is expected that one node will fail [15]. Thus runtime monitors that raise meaningful exceptions when protocols diverge from the behaviour specified by the session type can help improve fault tollerance.

We argue that, for Cloud providers, the performance overhead due to runtime monitors is low compared to the potential cost of problems avoided. In Cloud computing, it is perfectly acceptable to slow down transactions to guarantee correctness. For example, in Google Spanner [16,17] transactions observe a commit wait that delierately slows down transactions by a few milliseconds to guarantee globally meaningful commit timestamps.

3 Case Study: Protocols for a Cloud Coordinator

Our case study is a commercial Cloud provider, V3na, that provides integrated Software as a Service solutions for businesses. V3na provides a central access point to a portfolio of services, including document storage, document flow, and customer relations management. For comparison, market leading Cloud providers, such as Amazon or Rackspace, offer a portfolio of compute, storage and networking services that are exposed to users on demand. The central component in V3na is a Cloud coordinator that is responsible for exposing and integrating services that a user subscribes to, while managing user accounts and billing.

A typical scenario is when a user requires the document storage service. The user will first subscribe for the service either by registering to be billed or by entering a trial period. When the user has been successfully authenticated by the Cloud coordinator, requests to the API of the document store are delegated, by the Cloud coordinator, to the relevant document server for a renewable lease period. After delegation, the user interacts directly with the API of the document store until the session ends.

A major challenge was to automate the process of service integration as a reliable service. In particular, V3na implements protocols that address the following problems that can be addressed using sessions types:

- A customer can connect to a service for a trial period;
- A customer can connect to all services subscribed to through a single entry point;
- A subscription may be extended or frozen;
- Invoices and payment for use of services can be managed.

In this section, we illustrate a naive first implementation and a more scalable refinement of a protocol that implements the first scenario above.

3.1 First Attempt: Forwarding and Branching

We specify a first attempt of a simple protocol for connecting to a service. The protocol is informally specified as follows:

1. The user begins a session with Cloud coordinator and sends the request "connect to service" as a JSON message.
2. The Cloud coordinator selects either:
   (a) FAIL, if the user has no active session (not signed in).
   (b) OK, if the user has logged in and the request is validated.
3. If OK is selected, then, instead of responding immediately to the user, the Cloud initiates a new session with the relevant service. In the new session, the Cloud forwards the JSON message from the user to the service and receives a response from the service. The session between the Cloud and the service closes successfully.
4. Finally, the original session resumes and the Cloud forwards the response from the service to the user. From the perspective of the user it appears that the Cloud coordinator responded directly.

Protocol 1.1: User

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

Protocol 1.2: Cloud

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

Protocol 1.3: Service

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

Fig. 3: Protocol specifications for forwarding protocol.

In Figure 3, we provide the protocol specifications for each participant — the user, Cloud coordinator and service. The protocols between the user and the Cloud and between the Cloud and the service are dual, i.e. the specification of interaction from one perspective is opposite to the other perspective. SessionJ employs `outbranch` and `inbranch` operations to implement the branching behaviour. The `outbranch` operation is an internal choice, since the sender has control over the message sent. The `inbranch` operation is an external choice, since the receiver does not have control of the message received.

There is a fatal problem with the above protocol, from the perspective of a Cloud provider. The Cloud coordinator is involved in servicing all requests to services. As the number of services and users increases, the load on the Cloud coordinator will increase.

Soon, the Cloud coordinator will be unable to serve requests. The most basic economic advantage of Cloud computing, called elasticity [4], is that services can scale up and down to fulfil the demands of users. The above protocol cannot deliver elasticity.

3.2 Refined Protocol: Session Delegation and Iteration

We present a refined protocol that demonstrates iteration and session delegation. To avoid the Cloud coordinator becoming a bottleneck, the Cloud coordinator should delegate sessions to a service as soon as the user is authenticated for the service.

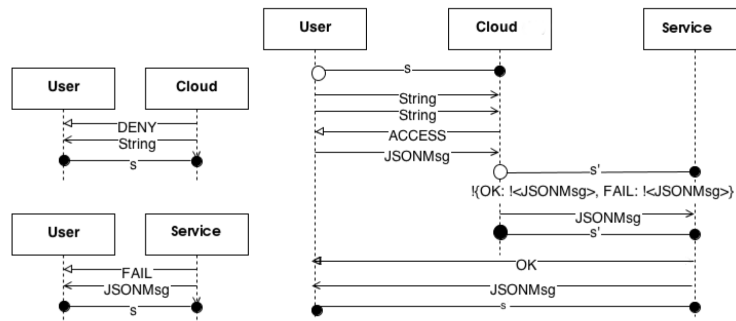

Fig. 4: Sequence diagram of interactions for delegation protocol.

Figure 4 depicts two related sessions s and s'. Session s begins with interactions between the user and the Cloud coordinator. However, after authentication, s' delegates the rest of session s from the Cloud coordinator to the service. Session s is completed by exchanging messages between the user and the service directly. We informally describe the global protocol in more detail:

1. The user begins a request session (session s in Fig. 4) with the Cloud coordinator.
2. The user logs in by providing the Cloud with a user name and password.
3. The Cloud coordinator receives the user credentials and verifies them. If the user is not authenticated and still has tries go back to step 2, otherwise continue.
4. If the user is not allowed to access the Cloud, the DENY branch is chosen and the session terminates. Otherwise, the ACCESS-branch is chosen and the session continues.
5. On the ACCESS branch, the user sends the connection request in a JSON message to the Cloud coordinator. The Cloud creates a new session with the service (session s' in Fig. 4). The new session delegates the remaining session with the user to the service, and also forwards relevant user request details to the service. Session s' is then terminated.
6. The service continues session s, but now interactions are between the user and the service. The service either responds to the user with OK or FAIL. In either case, the user receives the response directly from the service in a JSON message. Finally, session s is terminated.

Protocol 2.1: User

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

Protocol 2.2: Cloud

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

Fig. 5: User-Cloud interaction protocol specifications for delegation protocol.

In Figure 5, the user appears to interact with the Cloud coordinator. The iterative login, and first connection message is a direct interaction between the user and the Cloud coordinator. However, instead of the Cloud coordinator responding to the connection request, the session in Figure 6 is triggered.

Protocol 2.3: Cloud

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

Protocol 2.4: Service

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

Fig. 6: Cloud-Service interaction protocol specifications for delegation protocol.

The session in Figure 6 delegates the part of the session where the response OK or FAIL is selected by the service. This delegation is enabled by a higher order session type, where a socket of session type $!\{\mathtt{OK}: !\langle \mathit{JSONMessage} \rangle, \mathtt{FAIL}: !\langle \mathit{JSONMessage} \rangle\}$ is sent from the Cloud coordinator in protocol p_vs and received by the service in protocol p_sv. Following the delegation, a JSON message is sent from the Cloud coordinator to the service, which forwards on the relevant details of the user request.

Once the delegation has taken place, the service is able to complete the session that was begun by the Cloud coordinator. The service can negotiate directly with the user and either choose the OK branch or the FAIL branch, followed by sending the appropriate JSON message. For more complex scenarios, this simple choice between an OK and a FAIL message could be replaced by a more complex session between the user and the service.

The protocol presented in this section is scalable. The Cloud coordinator is only involved in authenticating users for access to services. The amount of data exchanged

during authentication is tiny compared to the amount of data exchanged by a service such as a document store.

3.3 Delegation Elsewhere: Payment for Services

Delegation is powerful elsewhere in the Cloud provider. At the end of each month, a user pays for the services used. The user may have multiple payment options. The two session types in Fig. 7 represent two different payment protocols. In the first protocol, the user pays with a credit card. In the second protocol, the user pays using a wallet, which is automatically recharged.

```
protocol p_payment {
  !<Goods>.?{
    VISA_MASTER: ?(CardDetails),
    TRANSFER: ?(TransferDetails)
  }.!{
    PAID: !<String>,
    DECLINED: !<String>,
    FAILED: !<String>
  }
}
```

```
protocol p_wallet {
  !<String>.?(Integer).?(Integer).!{
    PAYMENT_INACTIVE: !<OSMPMessage>,
    USER_NOT_FOUND: !<OSMPMessage>,
    OK: !<OSMPMessage>
  }
}
```

Fig. 7: Server side protocols for processing payments.

The user enters a session with the Cloud coordinator where, after authenticating, the payment option is selected then the payment is made. The session provided by the Cloud coordinator is presented on the left in Fig. 8 below.

```
protocol p_vu {
  begin.![
    ?(String).?(String)
  ]*.!{
    ACCESS: ?{
      PAYMENT: @p_payment,
      WALLET: @p_wallet
    },
    DENY: !<String>
  }
}
```

```
protocol p_vp {
  begin.!<String>.!<@p_payment>
}

protocol p_vw {
  begin.!<String>.!<@p_wallet>
}
```

Fig. 8: Delegating to chosen payment service.

However, the Cloud coordinator does not service either payment. One of the two delegation protocols on the right of Fig. 8 is invoked. The handling of the payment is delegated to either a bank or the wallet service within the Cloud provider. As in the previous example, delegation is performed by passing a higher order session type.

4 Future Work: Runtime Monitors and Intercloud Protocols

4.1 Language Independent Runtime Monitors

A limitation with the work presented is that services in a Cloud provider are not implemented exclusively in Java, or any other single language. The initial design of V3na was conducted using session types in SessionJ according to the methodology presented. However, as the start up company scales up to take on more clients, the development team is diversifying. The team now operates mainly in the Python based Django framework.

Session types can still be used by Python developers. Scribble [11] offers an alternative to SessionJ, where language independent runtime monitors [18] are statically checked according to session types. The runtime monitors dynamically check that low level communication patterns are within the space of behaviours specified by a session type. Scribble has already been used to monitor Python code in related work [19,20]. We argue that the approach offered by Scribble has a more promising future than SessionJ, since distributed systems are typically heterogeneous.

4.2 Session Types for Intercloud Protocols

For Cloud users, there are considerable benefits when applications can be hosted on more than one Cloud provider [4,21,22]. Users can build applications based on services provided by multiple Cloud providers. Furthermore, if data is replicated across multiple Cloud providers, customers can avoid becoming locked in to one provider. Thus customers are less exposed to risks such as fluctuations in prices and quality of service at a single provider. If a Cloud provider goes out of business, then customers entirely dependent on that Cloud provider also risk going out of business.

Several visions have been proposed for Intercloud protocols [1,23,24]. The main components debated for an Intercloud architecture are a *Cloud coordinator*, for exposing services, and a *Cloud broker* for mediating between Cloud coordinators. In this work, we have touched on some aspects of Cloud coordinators. The Cloud broker is a mediator that sits between the user and the Cloud coordinators for several Cloud providers. Another component debated is a *Cloud exchange*, which acts as a market place for services exposed by Cloud providers.

Based on our experience in this work, we suggest that session types are appropriate for specifying and correctly implementing protocols between Cloud coordinators and Cloud brokers. Like the delegation protocols between the Cloud coordinator and services, a Cloud broker will delegate communications to the Cloud coordinator as early as possible in the session. The protocols between Cloud brokers and Cloud coordinators are a critical component of the business model of a Cloud broker; hence, we argue that the overhead of deploying type checked runtime monitors is small compared to the potential risk posed by faults in protocols.

5 Conclusion

This case study addresses the question of whether session types have a role in Cloud computing. Competitive Cloud providers are looking for ways to better manage risks on

behalf of their customers. We argue that session types are one contribution that can help manage the risk posed by divergent critical components. In particular, we demonstrate that session types can be used to design, implement and verify protocols behind a Cloud coordinator that exposes services on demand to users.

Session type implementations such as SessionJ, as used in the work, and Scribble, as proposed for future work, involve some runtime monitoring. The runtime monitoring ensures that protocols stay within the space of behaviours permitted by a session type. We argue that the performance cost of dynamic runtime monitoring is small compared the risk managed. Divergent protocols can corrupt systems, while node failures are unavoidable at scale. Monitors can avoid divergence and help respond to node failures.

We found that session types provide an appropriate level of abstraction for quickly designing critical protocols. Session type implementations are accessible to programmers without background in formal semantics, including our industrial partners AlmaCloud Ltd. The level of abstraction provided by the SessionJ language, enabled effortless translation of business scenarios into verified implementations of protocols. We were able to refine our protocol from a simple forwarding protocol (Section 3.1) to a scalable delegation protocol (Section 3.2), due to support for higher-order message passing. The benefits of delegation are further highlighted by payment and wallet recharging transactions (Section 3.3).

We suggest that the methodology presented can be applied to emerging Intercloud protocols. In particular, protocols between Cloud brokers and Cloud coordinators delegate sessions similarly to protocols between Cloud coordinators and services. Furthermore, it is in the interest of Cloud brokers to minimise their exposure to risk due to divergent protocols or node failures. One approach to managing this risk is by using session types.

Acknowlegements. We thank the anonymous reviewers for their clear and constructive comments. We are particularly grateful to Ramesh Kini for his support for this project.

References

1. Buyya, R., Ranjan, R., Calheiros, R.N.: InterCloud: Utility-oriented federation of Cloud Computing environments for scaling of application services. In Hsu, C.H., Yang, L.T., Park, J.H., Yeo, S.S., eds.: ICA3PP (1). Volume 6081 of Lecture Notes in Computer Science., Springer (2010) 13–31
2. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The Eucalyptus open-source Cloud-Computing system. In Cappello, F., Wang, C.L., Buyya, R., eds.: CCGRID, IEEE Computer Society (2009) 124–131
3. Sotomayor, B., Montero, R.S., Llorente, I.M., Foster, I.T.: Virtual infrastructure management in private and hybrid Clouds. IEEE Internet Computing **13**(5) (2009) 14–22
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al.: A view of cloud computing. Communications of the ACM **53**(4) (2010) 50–58
5. Barros, A., Dumas, M., Oaks, P.: A critical overview of the Web services choreography description language. BPTrends Newsletter **3** (2005) 1–24

6. Carbone, M., Honda, K., Yoshida, N., Milner, R., Brown, G., Ross-Talbot, S.: A theoretical basis of communication-centred concurrent programming. WS-CDL working report, W3C (2006)
7. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for Web services. ACM Trans. Program. Lang. Syst. **34**(2) (2012) 8
8. Hu, R., Yoshida, N., Honda, K.: Session-based distributed programming in Java. In Vitek, J., ed.: ECOOP. Volume 5142 of Lecture Notes in Computer Science., Springer (2008) 516–541
9. Hu, R., Kouzapas, D., Pernet, O., Yoshida, N., Honda, K.: Type-safe eventful sessions in Java. In: ECOOP 2010–Object-Oriented Programming. Springer (2010) 329–353
10. Ng, N., Yoshida, N., Honda, K.: Multiparty Session C: Safe parallel programming with message optimisation. In Furia, C.A., Nanz, S., eds.: TOOLS (50). Volume 7304 of Lecture Notes in Computer Science., Springer (2012) 202–218
11. Honda, K., Mukhamedov, A., Brown, G., Chen, T.C., Yoshida, N.: Scribbling interactions with a formal foundation. In Natarajan, R., Ojo, A.K., eds.: ICDCIT. Volume 6536 of Lecture Notes in Computer Science., Springer (2011) 55–75
12. Basu, S., Bultan, T., Ouederni, M.: Deciding choreography realizability. ACM SIGPLAN Notices **47**(1) (2012) 191–202
13. Armstrong, J.: Getting Erlang to talk to the outside world. In: Proceedings of the 2002 ACM SIGPLAN workshop on Erlang, ACM (2002) 64–72
14. Simon Gay, M.H.: Subtyping for session types in the pi calculus. Journal Acta Informatica **42**(2-3) (2005) 191–225
15. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Communications of the ACM **51**(1) (2008) 107–113
16. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al.: Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation. USENIX Association (2012) 251–264
17. Ciobanu, G., Horne, R.: Non-interleaving operational semantics for geographically replicated databases. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, IEEE (2013) in press.
18. Bocchi, L., Chen, T.C., Demangeon, R., Honda, K., Yoshida, N.: Monitoring networks through multiparty session types. In Beyer, D., Boreale, M., eds.: FMOODS/FORTE. Volume 7892 of Lecture Notes in Computer Science., Springer (2013) 50–65
19. Neykova, R.: Session types go dynamic or how to verify your Python conversations. In: PLACES'13. Rome, Italy, 23 March. (2013) 34–39
20. Hu, R., Neykova, R., Yoshida, N., Demangeon, R., Honda, K.: Practical interruptible conversations - distributed dynamic verification with session types and Python. In Legay, A., Bensalem, S., eds.: RV. Volume 8174 of Lecture Notes in Computer Science., Springer (2013) 130–148
21. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation computer systems **25**(6) (2009) 599–616
22. Bernstein, D., Ludvigson, E., Sankar, K., Diamond, S., Morrow, M.: Blueprint for the Intercloud - protocols and formats for Cloud computing interoperability. In Perry, M., Sasaki, H., Ehmann, M., Bellot, G.O., Dini, O., eds.: ICIW, IEEE Computer Society (2009) 328–336
23. Cavalcante, E., Lopes, F., Batista, T.V., Cacho, N., Delicato, F.C., Pires, P.F.: Cloud integrator: Building value-added services on the cloud. In: NCCA. (2011) 135–142
24. Pawluk, P., Simmons, B., Smit, M., Litoiu, M., Mankovski, S.: Introducing STRATOS: A Cloud broker service. In Chang, R., ed.: IEEE CLOUD, IEEE (2012) 891–898