Language and Runtime Implementation of Sessions for Java

Raymond Hu[1], Nobuko Yoshida[1], and Kohei Honda[2]

[1] Imperial College London
[2] Queen Mary, University of London

1 Introduction

The purpose of this work is to incorporate the principles of *session types* into a concrete object-oriented language, specifically an extension of Java, as a basis for communications-based programming for distributed environments. Building on preceding theoretical studies of this topic, we present the first practical implementation of such a language, including a treatment of asynchronous communication, higher-order sessions (delegation) and session subtyping. This paper summarises the key design ideas of this work including the runtime architecture. Benchmark results for our current implementation demonstrate that our design introduces minimal overheads over the underlying transport, and is competitive with, in many cases outperforming, RMI, the standard API for typed inter-process communication in Java. A detailed version of this paper [12] is available online [11].

1.1 Background

Sockets Communication is becoming one of the central elements of application development in object-oriented programming languages. A frequent programming pattern in communications-based applications arises when processes interact via some *structured* sequence of communications, which as a whole forms the natural unit of a *conversation.* The *socket* (in particular, the TCP socket) is one of the most widely used programming abstractions for expressing such conversation structures. Available through APIs in many object-oriented languages such as Java and C$\sharp$, sockets represent the communication *endpoints* of a bidirectional byte stream abstraction, typically thought of as a *connection.* Although well suited to this purpose, socket-based programming suffers from several disadvantages.

- The byte stream abstraction is too low-level: no direct language level abstraction is provided for what each chunk of raw data means, let alone the structure of a conversation as a whole.
- Control flows in a pair of communicating processes should together realise a consistent conversation structure: the lack of an abstraction for the conversation as a whole means a programmer can easily fail to, for example, handle a specific incoming message or send a message when expected, with no way to detect such mistakes before runtime.

- The socket abstraction is directly coupled to a specific transport mechanism. Thus streams are tied to a physical connection, which complicates, for example, the delegation of an ongoing conversation.

These observations motivate the search for an abstraction mechanism for object-oriented programming that can naturally represent diverse conversation structures and be efficiently mapped to representative transport mechanisms, whilst preserving the familiar programming style of socket. Note Java RMI supports type-safe communication, but the rigid shape of method call makes it difficult to express general communication patterns using RPC.

Session Types A *session* is essentially a unit of sequential conversation, and the associated *session type* is an abstraction of the conversation structure and the messages exchanged, against which the communication behaviour of a program can be validated. Session types have been studied in many contexts in the last decade, including π-calculus-based formalisms [9]; multi-threaded functional languages [16]; CORBA [15]; operating systems [7]; and Web services [2].

With respect to the present problem, recent studies [6, 5, 4, 3] have demonstrated a clean integration of session type theory with object-oriented languages, through formalisms distilling selected object-oriented concepts for accurate analysis. Our work furthers these studies by contributing the design and implementation of a concrete, distributed language with session communication primitives and type system, including the key components of the runtime architecture such as the protocols that guarantee type-safe session delegation.

1.2 Problem Outline

The task at hand can be divided into three main problems.

Session programming abstractions. The abstractions should be expressive, enabling the representation of diverse conversation structures, and moreover *usable*, which stipulates a combination of clear syntax and intuitive (unsurprising) semantics, generating programming patterns natural to OOP. Naive implementation of the simplified theory has limitations other than usability, for instance the object calculi mentioned above [6, 5, 4, 3] do not permit operations on different sessions to be interleaved, precluding many real-world communication patterns. Exception handling for session operations is another such issue not addressed in the preceding theoretical work but certainly required for practical use.

Integration of session types. Session type theory has often focused on type inference, whereas Java has explicit type declaration: an implementation of session types closer to the latter would probably be easier for Java programmers to understand. Hence, syntax for session type declaration and an algorithm for static type checking (including new features such as interleaving and exceptions) are required. In addition, we consider how the standard imperative constructs in Java should combine with the chosen session programming abstractions, as reflected by our extended type system.

Runtime architecture. One of the main technical contributions of the present work is the design of the runtime support for asynchronous session communication, with the use of session type information as a fundamental element. Firstly, a safe execution of communication cannot be verified for distributed systems at compiletime, as a communicating party cannot statically assess the behaviour of peers discovered only at runtime. We solve this problem using a validation mechanism at session initiation, in which the two parties exchange their session types to determine whether or not their interactive behaviours are compatible. This session type information is used throughout the established session by both parties; for example, session types play a crucial part in coordinating the protocols for higher-order communication, *session delegation*. To validate the feasibility of our approach, we measure the performance of our primitives using several benchmark programs, comparing their cost over the untyped transport.

2 Approach

We outline the approach of our current design-implementation framework for the proposed language [11]. A core design feature is the use of session types in decoupling user description of session operations (abstraction concerns) and their execution mechanism (implementation concerns), analogous to high-level control flows and the underlying machine instructions in structured sequential programming. The key elements of our approach, addressing the issues described in the previous section, are as follows.

1. A **type syntax** for sessions based on [5, 6, 4, 3], but with enhanced readability and conformance to Java syntax.
2. Object-based **session programming primitives** that present an API-style interface. The fundamental abstraction is the *session-typed socket*, which represents a session endpoint.
3. A **new programming discipline/style**, derived from the first two points, for communications-based programs with guaranteed type and communication safety, which begins with a specification of intended communication structures using session types.
4. **Static session type checking**, implemented using the Polyglot compiler framework [14], coupled with **dynamic compatibility validation** at session initiation through a handshake protocol. Both components utilise session subtyping [8, 2].
5. The **runtime support** for the session abstraction, which encapsulates a variety of communication mechanisms with minimal overhead whilst abstracting from physical connections. The runtime incorporates the protocol for session initiation as well as delegation and close, and makes extensive use of session type information from point 3.

Session type declaration. The session type syntax abstracts the basic (object) send and receive actions, conditional behaviour through branching and selection (rather than binary if-statements, as in [5, 6, 4, 3]), and unbounded behaviour through while-loop iteration. We illustrate using a small example.

```
protocol p {
  begin.        // Session initiation
  ?[            // Iteration
    ?{            // Branch, two possible subconversations (labelled)
      GET: !(T),     // Send (object) type T
      PUT: ?(T)      // Receive (object) type T
    }
  ]*.
  end
}
```

T may itself be a session type, representing session delegation.

Session-typed sockets. We augment the standard socket to support the session communication primitives and session type checking. A session-typed socket, hereafter referred to as simply *session socket*, represents a session endpoint, over which session operations are performed like method calls to the socket object. We continue the above example.

```
STSocket s = STSocketImpl.create(p); // 'p' as declared above
s.request(host, port);                // begin.
T t = ...;
s.inwhile() {                        // ?[
  s.branch() {                         // ?{
    case GET: { s.send(t); }             // !(T),
    case PUT: { t = s.receive(); }       // ?(T)
  }                                    // }
}                                    // ]*.
s.close();                           //end
```

Whilst session programming is similar to standard API-based socket programming, the `branch` and `inwhile` operations (also, the corresponding `select` and `outwhile` operations), are new language constructs with intuitive semantics. The branch waits for the opposing session party to select a label, and inwhile iterates according to a control message implicitly communicated between the two session parties; these operations can be thought of as distributed versions of the standard switch and while statements that maintain synchronisation of control flow across both parties.

Session type checking. The type checker tracks the implementation of a session against the specified protocol, observing the correspondence between session operations and their types as demonstrated in the above example. For receive operations, both type inference, from the declared type, and checking, through explicit casting of the received object, are supported. The implementation may *subtype* the specification [8, 2]: a branch can offer more, and a select can use less, labels than specified. Type checking delegation operations is uniform with normal message types, but cannot occur within an iterative context.

The above issues relate to checking structural correspondence; we must also preserve session *linearity*. For example, aliasing of session sockets is forbidden, and session operations are not permitted within iterative constructs other than in/outwhile. Session implementation may diverge over conditional constructs provided there is a common supertype across all branches: the statement is typed as the lowest such type if it exists, for instance

```
// Session type: !{GET:?(T), PUT:!(T)}
if(...) {
  s.select(GET) { T t = s.receive(); } // !{GET:?(T)}
}
else {
  s.select(PUT) { s.send(new T()); }  // !{PUT:!(T)}
}
```

The type system allows session sockets to be passed as method arguments, which can be thought of as a "local" delegation: methods that accept session sockets specify the expected session type of the socket in place of `STSocket` in their declaration. We also have a treatment of exception handling for sessions.

```
try {
  s.request(host, port);
  ... // Body of session implementation
}
catch(SessionIncompatibleException sie) { ... }
catch(IOException ioe) { ... }
finally {
  try { s.close(); } catch (IOException x) {}
}
```

Sessions should be implemented using the try-catch construct to handle the listed exceptions (or within a method that throws these exceptions). The first exception signals that session initiation has failed because the opposing party has an incompatible behaviour for interaction; this is determined by a *duality* relation on session types (`!(T)` *duals* `?(T)`, etc.) that also permits subtyping e.g. a client that requires just one service may enter a session with a server offering several services. `IOException` is inherited from standard Java socket programming to signal communication failure during a session. For linearity, a session may not span multiple try-catch blocks unless delegated or passed as a method argument; thus, the occurrence of an exception necessarily terminates the session at both session parties. However, session interleaving is freely supported within a single try-statement, which expresses some semantic dependency between such sessions: an exception on any of the sessions is implicitly signalled to the others. Nested session exceptions can be thrown to an outer level, and the type checker permits only the close operation to be performed within the finally-block of a try-statement. The design of the session exception mechanism is ongoing work.

Runtime layer. The runtime layer encapsulates the underlying communication mechanisms; the interface to the runtime layer is the device by which session abstraction is decoupled from actual implementation. This enables exploitation of the transport, using session type information, for efficient communication with minimal overhead. The runtime is currently implemented in Java as the STSocket API: the compiler translates user code to the target API as a source-to-source translation. We describe some of the key components of the runtime.

- **Initiation handshake.** Session initiation involves a duality check between the session types of the two parties; if incompatible, the specified exception is raised at both parties. The current implementation uses literal class naming, which is sufficient for many examples. An earlier implementation [10] has support for class downloading; we plan to further investigate extensions that combine runtime verification of class compatibility and class downloading. Optimisations such as piggy-backing user messages on the handshake for short sessions are possible.
- **Delegation protocol.** The runtime incorporates an implementation of the delegation protocol [12], which governs the interaction between parties in bringing about session delegation in a transparent manner. Session types play a crucial role in treating asynchrony, one of the main design factors of the delegation protocol which includes the case for simultaneous (double) delegation of a session by both parties. Our protocol allows the delegating party to immediately close the delegated session, precluding (indefinite) message forwarding by a proxy agent (an alternative design).
- **Closing protocol.** An additional handshake at session closure is required to handle certain delegation cases, specifically when the passive party of a delegated session performs only output operations, namely send, select and outwhile. This is because this party may asynchronously complete his/her session contract before the delegating party is actually able to perform the delegation operation: the session must be kept "alive" until both parties agree that it has ended. The collaboration between the delegation and closing protocols is non-trivial, and to maintain asynchrony, the latter is performed in a separately spawned thread.

The current implementation of our work includes session sockets based on the Java Socket and NIO libraries. Performance results of several benchmark programs [12, 11] demonstrate that the session-based programming principles proposed in this work can be realised with low overhead. Indeed, our implementation in many cases exhibits better performance than RMI, the standard method for typed inter-process communication in Java. One factor is that the RMI supports additional features such as class downloading; however, the benchmark programs do not use this feature, and so the overheads incurred by RMI are minimal for this point. Moreover, the presence of session types and the information they convey, such as communication direction and (bounded) message size, suggest the potential for further optimisation at both the user and transport level.

3 Conclusion and Related Work

The present work clarifies, through a concrete implementation, the significant impact that the introduction of sessions and session types into object-programming languages can have, on both programming discipline and runtime architecture. Below we summarise the key technical contributions of the present work over the preceding (mainly theoretical) work on session types.

- **Practical programming methodology for session types.** Starting from protocol declaration, we ensure type safety through combined type checking and inference, extending typability with session subtyping.
- **Integration of session types into OOP.** This is realised by the design of session sockets, extending the type system to prevent aliasing of the socket objects, and through a natural and consistent integration with standard imperative constructs and exception handling, allowing session interleaving.
- **Runtime architecture.** The design of the runtime mechanisms is key to the practical use of session types, which in turn are a fundamental element of this design. The runtime encapsulates the underlying message transport and session communication protocols, including the session initiation handshake, where session types are exchanged and validated; and the delegation protocol, which separates the session abstraction from physical connections.

In the following we discuss some of the related works; a more complete discussion such as a comparison with several typed languages based on process calculi (Pict, Polyphonic C♯, Cω, the Concurrency and Coordination Runtime (CCR), JavaSeal, Occam-pi and X10) can be found in [12].

One of the first applications of session types in practice is found in Web Services. The Web Service Description Language (WS-CDL) [17], developed by a W3C standardisation working group, employs a variant of session types to address the need for static validation of business protocols. A WS-CDL description is realised as the interactions of distributed endpoints written in languages such as Java or WS4-BPEL [1], or that proposed by the present work. Recent work [2] has studied the principles behind deriving sound and complete endpoint implementations from a CDL description.

A variant of session types has been combined with a derivative of C♯ for systems programming, playing a crucial role in the development of Singularity OS for shared memory uni/multiprocessor environments [7]. Session types, referred to as contracts, are used to specify the interaction between OS modules as message-passing conversations. Reflecting the hardware and software assumptions of this work (i.e. shared memory and homogeneity of OS modules), features for distribution, such as the session initiation handshake, are not significant. Further, their work does not support subtyping, another requirement of practical distributed applications, and promotes a programming methodology more tightly coupled to the underlying execution mechanism than our approach.

An implementation of a session type system in Haskell is studied in [13] through an encoding of a simple session calculus. Again, this work is targeted at a concurrent, but not distributed, environment. It may be difficult to realise the session compatibility check or type-safe delegation since the encoding does not directly type I/O channels.

References

1. WS-BPEL OASIS Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.
2. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP '07*, LNCS. Springer-Verlag, 2007.
3. Mario Coppo, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. To appear in FMOOSE'07. http://www.di.unito.it/ dezani/papers/cdy.pdf, 2007.
4. Mariangiola Dezani-Ciancaglini, Sophia Drossopoulou, Elena Giachino, and Nobuko Yoshida. Bounded Session Types for Object-Oriented Languages. Submitted for publication. http://www.di.unito.it/ dezani/papers/ddgy.pdf, 2007.
5. Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP '06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.
6. Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. A Distributed Object Oriented Language with Session Types. In *TGC '05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.
7. Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen C. Hunt, James R. Larus, , and Steven Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys 2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.
8. Simon Gay and Malcolm Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
9. Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language Primitives and Type Discipline for Structured Communication-Based Programming. *LNCS*, 1381:122–??, 1998.
10. Raymond Hu. Implementation of a Distributed Mobile Java. Master's thesis, Imperial College London, 2006.
11. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Online appendix of this paper. http://www.doc.ic.ac.uk/ rh105/sessiondj.html.
12. Raymond Hu, Nobuko Yoshida, and Kohei Honda. Type-safe Communication in Java with Session Types (18pp. draft). March 2007.
13. Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer-Verlag, 2004.
14. Polyglot home page. http://www.cs.cornell.edu/Projects/polyglot/.
15. Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the Behavior of Objects and Components using Session Types. In *FOCLASA '02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.
16. Vasco T. Vasconcelos, António Ravara, and Simon Gay. Session Types for Functional Multithreading. In *CONCUR '04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.
17. Web Services Choreography Working Group. Web Services Choreography Description Language. http://www.w3.org/2002/ws/chor/.