SCISPACE formerly Typeset

Open access • Book Chapter • DOI:10.1007/978-3-540-70592-5_22

Session-Based Distributed Programming in Java — Source link

Raymond Hu, Nobuko Yoshida, Kohei Honda

Institutions: Imperial College London, Queen Mary University of London

Published on: 07 Jul 2008 - European Conference on Object-Oriented Programming

Topics: Session (computer science), Message passing, Delegation (computing), Syntax (programming languages) and Java

Related papers:

- Language Primitives and Type Discipline for Structured Communication-Based Programming
- Multiparty asynchronous session types
- An Interaction-based Language and its Typing System
- Subtyping for session types in the pi calculus
- Session types for object-oriented languages

Share this paper:

View more about this paper here: https://typeset.io/papers/session-based-distributed-programming-in-java-1loiuy32ww

Session-Based Distributed Programming in Java

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

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

Abstract. This paper demonstrates the impact of integrating session types and object-oriented programming, through their implementation in Java. Session types provide high-level abstraction for structuring a series of interactions in a simple and concise syntax, and ensure type-safe communications between distributed peers. We present the first full implementation of a language and runtime for session-based distributed programming featuring asynchronous message passing, delegation and session subtyping, combined with class downloading and exceptions. The compilation and runtime framework of our language effectively maps session abstraction onto underlying transports and guarantees communication safety through static and dynamic session type checking. We have implemented two alternative protocols for performing correct and efficient session delegation and prove their correctness. Benchmark results show the session abstraction can be realised with minimal runtime overhead.

1 Introduction

Communication in object-oriented programming. Communication is becoming a fundamental element of application development. Web applications increasingly combine the use of numerous distributed services; an off-the-shelf CPU will soon host thousands of cores per chip; corporate integration builds complex systems that communicate using standardised business protocols; and sensor networks will place hundreds of processing units per square meter. A frequent pattern in the communications-based programming involves processes interacting via some structured sequence of communications, which as a whole form a natural unit of *conversation*. In addition to basic message passing, a conversation may involve repeated exchanges or branch into one of multiple paths. Structured conversations of this nature are ubiquitous, arising naturally in server-client programming, parallel algorithms, business protocols, Web Services, and application-level network protocols such as SMTP and FTP.

Objects and object-orientation are a powerful abstraction for sequential and shared variable concurrent programming. However, objects do not provide sufficient support for high-level abstraction of distributed communication and conversations, even with a variety of communication API supplements. Remote Method Invocation (RMI), for example, cannot directly capture arbitrary conversation structures; interaction is limited to a series of separate send-receive exchanges. More flexible interaction structures can, on the other hand, be expressed through lower-level (TCP) socket programming, but communication safety is lost: raw byte data communicated through sockets is inherently untyped and conversation structure is not explicitly specified. Consequently, programming errors in communication cannot be statically detected with the same level of robustness as standard type checking protects object type integrity.

The study of *session types* has explored a type theory for structured conversations in the context of process calculi [30, 16, 18] and a wide variety of formal systems and programming languages. A session is a conversation instance conducted over, logically speaking, a private channel, isolating it from interference, and a session type is a specification of the structure and message types of the conversation as a complete unit. Unlike method call, which implicitly builds a synchronous, sequential thread of control, communication in distributed applications is often interleaved with other operations and concurrent conversations. Sessions provide a high-level programming abstraction for such communications-based applications, grouping multiple interactions into a logical unit of conversation, and guaranteeing their communication safety through types.

Challenge of session-based programming. This paper demonstrates the impact of integrating session types and object-oriented programming in Java. Preceding works include theoretical studies of session types for object-oriented calculi [13, 11], and the implementation of a systems-level object-oriented language with session types for shared memory concurrency [14]. We further these works by presenting the first full implementation of a language and runtime for session-based distributed programming featuring asynchronous message passing, delegation and session subtyping, combined with class downloading and exceptions. The design and implementation of such a language poses several non-trivial technical challenges: the following summarises the key problems and how they are addressed in the compilation and runtime framework of our language.

1. *Integration of object-oriented and session programming disciplines.* We extend Java with minimal syntax for session types and structured communication operations. Session-based concurrent and distributed programming involves specifying the intended communication protocols using session types, implementing these protocols using the session operations, and static verification of the session implementations against the protocol specifications. Session types describe interfaces for conversation in the way Java interfaces describe method-call interaction.
2. *Ensuring communication safety for distributed applications.* Communication safety is guaranteed through a combination of static and dynamic session type checking. Static verification of session implementations ensures that each session party conforms to the local protocol specifications; runtime verification at session initiation checks that the session parties implement compatible protocols.
3. *Supporting session abstraction over concrete transports.* The compilation-runtime framework of our language maps application-level session operations, including delegation, to runtime communication primitives, which can be implemented over a range of concrete transports; our current implementation uses TCP. Benchmark results show that session communication incurs little runtime overhead over the underlying transport.

A key technical contribution of our work is the implementation of distributed session delegation: transparent, type-safe endpoint mobility is a defining feature

that raises the session abstraction above the underlying transport. We have implemented two alternative protocols for coordinating the actions required to perform a delegation, and we prove their correctness and discuss their design trade-offs. Our work also demonstrates how the integration of session types and objects can support runtime features such as eager remote class loading and eager class verification for message type-safety.

Outline. Section 2 illustrates the key features of session programming. Section 3 describes the design elements of the compilation and runtime framework. Section 4 discusses the implementation of session delegation and its correctness. Section 5 presents benchmark results comparing session-based programs against equivalent implementations that use Java Socket and RMI. Section 6 discusses related work and Section 7 concludes. The compiler and runtime, program examples and omitted details are available at [21].

2 Programming with Sessions

This section illustrates the central ideas of programming in our session-based extension of Java, which we call SJ for short, by working through an example, an online ticket ordering system for a travel agency. This example comes from a web service usecase in WS-CDL-Primer 1.0 [4], designed to capture typical collaboration patterns found in many business protocols [6, 8, 31]. Figure 1 depicts the interaction between the three parties involved, a client (Customer), the travel agency (Agency) and a travel service (Service). Customer and Service are initially unknown to each other but later communicate directly (transparently to Customer) through the use of delegation, to enable dynamic mobility of sessions in a type-safe manner. The overall conversation scenario proceeds as follows.

1. Customer begins an *order session* s with Agency, then requests and receives the price for the desired journey. This exchange may be repeated an arbitrary number of times for different journeys under the direction of Customer.
2. Customer either accepts an offer from Agency or decides that none of the received quotes are satisfactory; the two possible paths are illustrated separately as adjacent flows in the diagram.
3. If an offer is accepted, Agency opens the session s' with Service and *delegates* the remaining interaction for s with Customer to Service. The particular travel service to which Agency connects is likely to depend on the journey chosen by Customer, but this logic is external to the present example.
4. Customer then sends a delivery address (unaware that he/she is now talking to Service), and Service replies with the dispatch date for the purchased tickets. The transaction is now complete.
5. Customer cancels the transaction if no quotes were suitable and the session terminates.

The rest of this section describes how this application can be implemented in SJ. Roughly speaking, session programming consists of two steps: specifying the intended communication protocols using session types, and implementing these protocols using session operations.

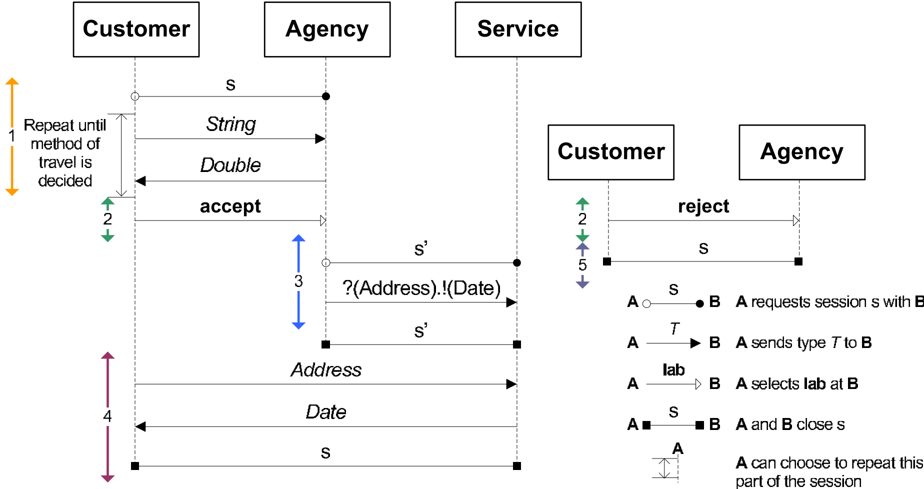


Fig. 1. A ticket ordering system for a travel agency.

Protocol specification. Protocols are specified through session type declarations. We call the protocol (i.e. session type) for the order session between Customer and Agency, from Customer's side, `placeOrder`, and the protocol of the *dual* type from Agency's side, `acceptOrder`; both are specified below.

```
protocol placeOrder {
  begin. // Commence session.
  ![     // Can iterate:
     !<String>. // send String
     ?(Double)  // receive Double
  ]*.
  !{   // Select one of:
     ACCEPT: !<Address>.?(Date),
     REJECT:
  }
}
```

Order protocol: Customer side.

```
protocol acceptOrder {
  begin.
  ?[
     ?(String).
     !<Double>
  ]*.
  ?{
     ACCEPT: ?(Address).!<Date>,
     REJECT:
  }
}
```

Order protocol: Agency side.

Session types are declared using the `protocol` keyword. Let us focus on the left-hand side, `placeOrder`: it first says a conversation will repeat as many times as desired (denoted by `![..]*`), a sequence which consists of sending a String (`!<String>`) and receiving a Double (`?(Double)`). Customer then selects (`!{...}`) one of the two options, `ACCEPT` and `REJECT`. If `ACCEPT` is chosen, Customer sends an Address and receives a Date, then the session terminates; if `REJECT`, the session terminates immediately. `acceptOrder` is the session type *dual* to `placeOrder`, given by inverting the input `!` and the output `?` symbols in `placeOrder`.

Session sockets. After declaring the intended communication protocols, the next step is to create session sockets for opening sessions and performing session operations. There are three entities:

– *Session server socket* of type `SJServerSocket`, which listens for session requests and accepts those that are compatible;
– *Session server address* of type `SJServerAddress`, which gives the address of a session server socket and the type of session it accepts; and
– *Session socket* of type `SJSocket`, which represents one endpoint of a session channel and is used to request a session from a server.

SJ uses the terminology from standard socket programming for familiarity. The session sockets and server sockets correspond to their standard socket equivalents, sockets and server sockets, enhanced with associated session types to guarantee communication safety. Session server sockets accept a request if the type of the server is compatible with the requesting socket; the server will then create a fresh session socket for the new session. A session server address is the client-side channel to a session server socket and is specified by the IP address and TCP port of a session server socket together with its session type seen from the client. Server addresses can be communicated by sessions to other parties who may then request a new session with the server. Once a session is established messages are exchanged through a session socket, from one end of the session to the other end and back. The occurrences of a session socket in a SJ source program clearly signify the thread of a conversation among other commands.

Session server sockets. Parties that offer session services, like Agency, use a session server socket to accept session requests.

```
SJServerSocket ss_ac = SJServerSocketImpl.create(acceptOrder,port);
```

The server party processes a session request by

```
s_ac = ss_ac.accept();
```

where `s_ac` is an uninitialised SJSocket reference. As for standard server sockets, the `accept` operation blocks until a session request is received, but when one arrives, the server socket additionally validates that the type of the requested session is compatible with that offered by the server, see § 3 for the details.

Session server address and session sockets. The creation of a session server address registers, at the session type level, that the server at the specified address can accept sessions of the specified type. At the client side we set:

```
SJServerAddress c_ca = SJServerAddress.create(placeOrder, host, port);
```

which allows a client to initiate a conversation with the server. The server address is typed with the session type seen from the client side, in this case `placeOrder`. As we shall see later this type allows this address to be communicated safely. Customer also uses `c_ca` to create a session socket:

```
SJSocket s_ca = SJSocketImpl.create(c_ca);
```

and request a session with Agency by

```
s_ca.request();
```

Assuming the server socket identified by `c_ca` is open, `request` blocks until Agency performs the corresponding accept. Then the requesting and accepting sides independently validate session compatibility and if validated the session between Customer and Agency is established. If incompatible, an exception is raised at both parties (see Session Failure later).

Session communication (1): send and receive. After the session has been established, the session socket `s_ca` belonging to Customer (respectively `s_ac` for Agency) is used to perform the actual session operations according to the session type specification of `placeOrder`. Static session type checking ensures that this contract is obeyed with allowance for session subtyping, see § 3.2 for the details.

The basic message passing operations, performed by `send` and `receive`, communicate typed objects; communication is asynchronous. The opening exchange of `placeOrder` directs Customer to send the details of the desired journey, `!<String>`, and receive a price quote, `?(Double)`.

```
s_ca.send("London to Paris, Eurostar"); // !<String>.
Double cost = s_ca.receive(); // ?(Double)
```

Session communication (2): iteration. Iteration is abstracted by the two mutually dual types written `![...]*` and `?[...]*` [13, 11]. Like regular expressions, `[...]*` expresses that the interactions in `[...]` may be iterated zero or more times; the prefix `!` indicates which party controls the iteration, and the `?` peer follows this decision. These types are implemented using the `outwhile` and `inwhile` [13, 11] constructs, which can together be considered a distributed version of the standard while-statement. The opening exchange of `placeOrder`, `![!<String>.?(Double)]*`, may be repeated under the decision of Customer: the implementation of this type and its dual are as follows.

```
boolean decided = false;
... // Set journey details.
s_ca.outwhile(!decided) {
 s_ca.send(journDetails);
 Double cost = agency.receive();
 ... // Set decided to true or
 ... // change details and retry
}
```

```
s_ac.inwhile() {
  String journDetails
             = s_ac.receive();
  ... // Calculate the cost.
  s_ac.send(price);
}
```

Like the standard while-statement, the `outwhile` operation evaluates the boolean condition for iteration, `!decided`, to determine whether the loop continues or terminates. The key difference is that this decision is implicitly communicated to the session peer (in this example from Customer to Agency), synchronising the control flow between two parties.

Agency is programmed with the dual behaviour. Note `inwhile` does not have a conditional expression: this is because the decision to iterate or exit is made by Customer, communicated to Agency at each iteration. These primitives for iterative communication can substantially reduce the design complexity of conversation, compared to direct implementation using I/O stream.

Session communications (3): branching. A session may branch down one of multiple conversation paths leading to differing sub-conversations. In `placeOrder` we have `!{ACCEPT: !<Address>.?(Date), REJECT: }`: hence if Customer, who is the `!` party, selects `ACCEPT` then the session proceeds into a sub-conversation with two communications; selecting `REJECT` immediately terminates the session.

Agency, according to the `?{ACCEPT: ?(Address).!<Date>, REJECT: }` type in `acceptOrder`, has the dual behaviour; as the `?` party, Agency must wait for Customer to make the branch decision. Naturally Agency must support all possible paths although Customer will only select one when the session is executed.

The branch types are implemented using the `outbranch` and `inbranch`. This pair of operations can be considered a distributed switch-statement, or one can view `outbranch` as something similar to method invocation, with `inbranch` representing the target object waiting with one or more methods. We continue the example with the next part of the programs for Customer and Agency.

```
if(want to place an order) {
  s_ca.outbranch(ACCEPT) {
    s_ca.send(address);
    Date dispatchDate = s_ca.receive();
  }
}
else { // Don't want to order.
  s_ca.outbranch(REJECT) { }
}
```

```
s_ac.inbranch() {
  case ACCEPT: {
    ...
  }
  case REJECT: { }
}
```

The condition of the if-statement in Customer (whether or not Customer wishes to purchase tickets) determines which branch will be selected at runtime. The body of `ACCEPT` in Agency is completed in Session Delegation below.

Session failure. Sessions are implemented within session-try constructs as:

```
try (s_ac, ...) {
  ... // Implementation of session 's_ac' and others.
} catch (SessionIncompatibleException sie) {
  ... // One of the above sessions could not be initiated.
} catch (SessionIOException ioe) {
  ... // I/O error on any of the above sessions.
} finally { ... } // Optional.
```

The session-try is extended from the standard Java try-statement to ensure that session implementations, which may be freely interleaved, are completed within the specified scope. Sessions may fail at initiation due to incompatibility, or generally at any point due to I/O error. Failure is signalled through the propagation of terminal exceptions: failure of one session will implicitly fail all other sessions within the same scope. Any other exceptions that cause the flow of control to leave the session-try before all sessions have been completed will also fail those sessions. However, a party that has successfully completed its side of a session may asynchronously leave its session scope. Nested session-try statements offer programmers the choice to fail the outer session if the inner session fails

or to consume the exception and continue. Operations on outer sessions may not be performed within a nested scope, and nested scopes are completed before exceptions on outer sessions are processed.

Session delegation. If Customer is happy with one of Agency's quotes it will select `ACCEPT`. This causes Agency to open the delegation session with Service and delegate to it the remainder of the conversation with Customer, as specified in `acceptOffer`. After the delegation, Agency relinquishes the session and Service agrees to complete it: this ensures that the contract of the original order session will be fulfilled. At the application-level, the delegation involves only Agency and Service; Customer will proceed to interact with Service unaware that Agency has left the session, which is evidenced by the absence of any such action in `placeOrder`. The session between Agency and Service is specified by:

```
protocol delegateSession {
  begin.!<?(Address).!<Date>>
}
```

```
protocol receiveSession {
  begin.?(?(Address).!<Date>)
}
```

Delegations are abstracted as *higher-order session types* [13, 18, 11], where the specified message type is itself a session type; in this example, `?(Address).!<Date>`. The message type denotes the remainder of the protocol for the session being delegated at the point of delegation; the party that receives the session will resume the conversation from that stage.

In the code materialising this type, delegation is naturally represented by a sending action of the session socket of the session to be delegated. Continuing our example, Agency can delegate the order session with Customer to Service by

```
case ACCEPT: {
  SJServerAddress c_as = ... // Specify delegateProtocol.
  SJSocket s_as = SJSocketImpl.create(c_as);
  s_as.request();
  s_as.send(s_ac); // Delegation. Agency has finished with s_ac.
}
```

and Service receives the delegated session from Agency by

```
SJServerSocket ss_sa = SJServerSocketImpl.create(receiveSession, port)
SJSocket s_sa = service.accept();
SJSocket s_sc = s_sa.receive(); // Receive delegated session.
```

Service then completes the session with Customer.

```
Address custAddr = s_sc.receive();
Date dispatchDate = ... // Calculate dispatch date.
s_sc.send(dispatchDate);
```

The implementation of session delegation in the SJ runtime uses protocols that coordinate the involved session peers, and is explained in detail in § 4.

The above example illustrates only the basic features of our language. The source code of this example is available at [21] together with compiler and runtime, along with other SJ-programs that feature more complex interactions, including implementation of business protocols from [6, 5].

```
begin       c.request(), ss.accept()        // Session initiation.
!<C>        s.send(obj)                     // Object 'obj' is of type C.
!<S>        s1.send(s2)                     // 's2' has remaining contract S.
?<T>        s.receive()                     // Type inferred from protocol.
!{L:T,..}   s.outbranch(L){...}             // Body of case L has type T, etc.
?{L:T,..}   s.inbranch(){...}               // Body of case L has type T, etc.
![T]*       s.outwhile(boolean expr){...}   // Outwhile body has type T.
?[T]*       s.inwhile(){...}                // Inwhile body has type T.
```

Fig. 2. Session operations and their types.

3 Compiler and Runtime Architecture

3.1 General Framework

The compilation-runtime framework of SJ works across the following three layers. This section describes how session type information plays a crucial role in each of these layers.

Layer 1 SJ source code.
Layer 2 Java translation and session runtime APIs.
Layer 3 Runtime execution: JVM and Java libraries.

Through this framework, transport-independent session operations at the application-level are compiled into more fine-grained communication actions on a concrete transport. Layer 1 is mapped to Layer 2 by the SJ compiler: session operations are statically type checked and translated to communication primitives supported by the session runtime interface. Layer 3 implements the session runtime interface over a concrete transport and performs dynamic session typing. The compiler comprises approximately 6KLOC of Java extension to the base Polyglot framework [26]. The current implementation of the session runtime uses TCP and consists of approximately 1 KLOC of Java.

A core principle of this framework is our view that explicit declaration of conversation structures, coupled with the static and dynamic type checking of SJ, provides a basis for well-structured communication programming. We envisage programmers working on a distributed application first agree on the protocols through which the components interact, specified using session types, against which the implementation of each component is statically validated. Dynamic type checking then ensures that session peers implement compatible protocols (i.e. the components have been correctly composed) before sessions are commenced during runtime. The mechanisms encapsulated by the session runtime by which the session operations (initiation, send/receive, branch, loop, delegation) and additional features (eager class downloading, eager class verification) are performed are discussed in § 3.3 and § 4.

3.2 The SJ Compiler and Type-checking

The SJ compiler type-checks the source code according to the constraints of both standard Java typing and session typing and, using this type information, maps

the SJ surface syntax to Java and the session runtime APIs. Type-checking for sessions types starts from validating the linear usage of each session socket, preventing aliasing and any potential concurrent usage. On the basis of linearity, the type-checker ensures that each session implementation conforms to the specified protocol with respect to the communicated message types and conversation structure, as stipulated by the correspondence between session operations and type constructors given in Figure 3.

Some subtle typing cases arise when session sockets are passed as method arguments (a consequence of integrating session and object types) and for session delegation; both involve transfer of responsibility for completing the remainder of session being passed. Methods that accept session socket arguments must declare the expected session types:

```
int foobar(protocol p s, ...) throws SessionIOException {
      ... // Implementation of s according to p.
}
```

Session passing is also subject to linearity constraints. For example, the same session socket cannot be passed as more than one parameter in a single method call, and the following examples are similarly badly typed since they delegate the session `s2` multiple times.

```
while(...) { s1.send(s2); }
```

```
s1.inwhile() { s1.send(s2); }
```

Session subtyping is an important feature for practical programming, permitting message type variance for send and receive operations and structural subtyping for branching [17, 9]. Message type variance follows the object class hierarchy, again from the integration of session and object types; intuitively, a subtype can be sent where a supertype is expected. Structural session subtyping [17, 8] has two related purposes. Firstly, an outbranch implementation needs only select from a subset of the cases offered by the protocol, and vice versa for inbranch; secondly, inbranch (server) and outbranch (client) types are compatible at runtime if the former supports a superset of the all cases that the latter may require. The following demonstrates the various kinds of session subtyping.

```
protocol thirstyPerson {
  begin.!{
     COFFEE: !<Euros>,
     TEA: !<Euros>
  }
}
...
if (...) { // Implemented...
  s.outbranch(COFFEE) { ... };
}
else { // ...a coffee addict.
  s.outbranch(COFFEE) { ... };
}
```

```
protocol vendingMachine {
  begin.?{
     COFFEE: ?(Money),
     TEA: ?(Money),
     CHOCOLATE: ?(Money)
  }
}
...
s.inbranch() {
  case COFFEE: { ... }
  case TEA: { ... }
  case CHOCOLATE: { ... }
  case CAT: { ... }
} // Weird, but type-safe.
```

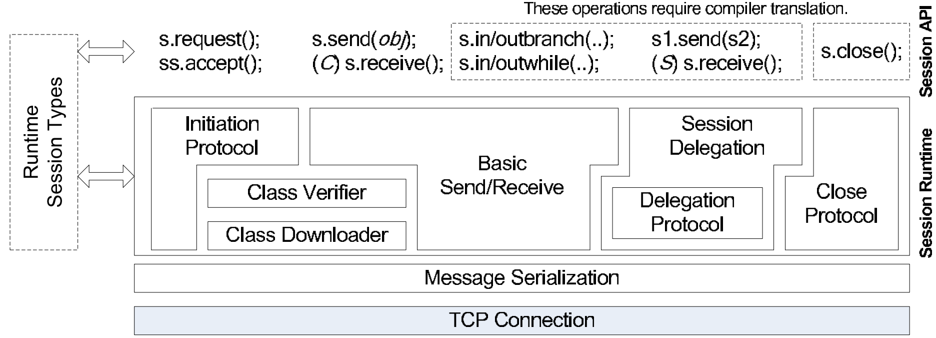


Fig. 3. The structure of the SJ session runtime.

This example uses subtpying at both compilation and runtime. Message subtyping for sessions is augmented by remote class loading, discussed in § 3.3.

3.3 The Session Runtime

The overall structure of the session runtime is illustrated in Figure 3. The SJ compiler translates the session operations, the upper-most layer of the figure, into the communication primitives of the session APIs. The session runtime implements these primitives through a set of interdependent protocols that model the semantics of the translated operations; these protocols map the abstract operations to lower-level communication actions on a concrete transport. In this way, the session APIs and runtime cleanly separate the transport-independent session operations from specific network and transport concerns. The main components as depicted in the above figure are:

- the *initiation protocol*, *class verifier* and *class downloader*;
- *basic send/receive*, which also supports in/outbranch and in/outwhile;
- the *delegation protocol* and *the close protocol*.

Session type information, tracked by the runtime as each session operation is performed, is crucial to the operation of several of the component protocols. The following describes each component except delegation discussed in § 4.

Basic send/receive. The properties of session types, such as strict message causality for session linearity, require the communication medium to be reliable and order preserving. This also allows the session branch and while operations to be implemented by sending/receiving a single message, i.e. the selected branch label or loop condition value. As depicted in the above figure, our current implementation uses TCP: each active session is supported by a single underlying TCP connection. However, the session runtime would not require much modification to use other transports with the above properties, such as SCTP. Session messages, either Java objects or primitive types, are transmitted through the TCP byte stream using standard Java serialisation. Basic send/receive is asynchronous, meaning a basic send does not block on the corresponding receive; however, receive does block until a (complete) message has been received.

Session initiation and dynamic type checking. Session initiation takes place when the `accept` and `request` operations of a server and client reduce. The initiation protocol establishes the underlying TCP connection and then verifies session compatibility. The two parties exchange the types of the sessions they implement, which have been statically verified, and each independently validates compatibility allowing for session subtpying. If successful, then the session has been established, otherwise both parties raise an exception and the session is aborted. The initiation protocol can also perform eager class downloading and/or eager class verification, depending on the parameters set on each session socket.

Class downloader and class verifier. Our runtime supports a remote class loading feature similar to that of Java RMI. Remote class loading is designed to augment message subtyping, enabling the communication of concrete message types that implement the abstract types of a protocol specification. Session peers can specify the address of an external HTTP class repository (codebase) from which additional classes needed for the deserialization of a received message object can be retrieved. By default, remote class loading is performed *lazily*, i.e. at the time of deserialization, as in RMI. Alternatively, a party may chose to *eagerly* load, at session initiation, all possibly needed classes as can determined from the session type of the session peer (although, due to subtyping, lazy class loading may still be required during the actual session). Similarly, session peers may chose to perform eager class verification for all shared classes at session initiation; class verification is implemented using the standard `SerialVersionUID` checks for serializable classes.

The close protocol. SJ does not have an explicit session close operation; instead a session is implicitly closed when program flow leaves the enclosing session-try scope. There are essentially three ways for this to happen. The first case is when both parties finish their parts in a conversation and the session terminates normally. The second is when an exception is raised at one or both sides, signalling session failure. In this case, the close protocol is responsible for propagating the exception to all other active sessions within the same scope, maintaining consistency across such dependent sessions. The third more subtle case arises due to asynchrony: it is possible for a session party to complete its side of a session before or whilst the peer is performing a delegation. Section 4 discusses how the delegation and close protocols interact in such case.

4 Protocols for Session Delegation

Session delegation is a defining feature of session-typed programming; transparent, type-safe endpoint mobility raises the session abstraction above ordinary communication over a concrete transport. This means a conversation should continue seamlessly regardless of how many times either party changes location, at any point of the conversation. Consequently, each session delegation involves intricate coordination between three or even four parties, if both peers simultaneously delegate the same session. In SJ, delegation is performed using the same

syntax as the ordinary `send` and `receive`; the compilation-runtime framework of SJ resolves delegation using statically verified session type information.

This section, after examining the trade-offs between general implementation strategies, presents two alternative runtime protocols for delegation, and outlines their correctness arguments. Henceforth we call the parties involved in a delegation the *s-sender* (for session-sender) and the *s-receiver* (for session-receiver), and the peer of the s-sender for the session being delegated the *passive-party*. We discuss the implementation of delegation in the context of a TCP-like connection-based transport in accordance with our implementation of the SJ runtime.

4.1 Design Options for the Delegation Protocol

Indefinite redirection and direct reconnection. One way to implement delegation is for the s-sender to indefinitely redirect all communications of the session, in both directions, between the s-receiver and passive-party, in a similar scheme to Mobile IP [22]. The merit of this approach is that no action is required on the part of the passive-party. At the same time, communication overhead for the indirection can be expensive, and the s-sender is needed to keep the session alive even after it has been logically delegated. Thus, the s-sender is prevented from terminating even if it has completed its own computation, and its failure also fails the delegated session. These observations suggest this design option is unsuitable for dynamic network environments such as ubiquitous computing and P2P networks.

An alternative design is to directly reconnect the underlying transport connections to reflect the conversation dynamics: we first close the original connection for the session being delegated, and then replace it with a connection between the new session peers (the s-receiver and the passive-party). While this demands additional network operations on the part of the passive-party, it frees the s-sender from the obligation to indefinitely take care of the delegated session, giving a robust and relatively efficient solution. Although indefinite redirection is still relevant for fixed and reliable hosts, we believe that direct reconnection has overwhelming functional advantages.

Challenge of reconnection-based design. The design of a delegation protocol based on reconnection raises an interesting problem due to asynchrony of communication (send is non-blocking). We explain by returning to the example in § 2. As specified in the global description of application (Figure 1), if Customer selects `ACCEPT`, Agency delegates the active order session with Customer to Service, and then Customer should send the `Address` to Service. In practise, however, Customer is operating in parallel with Agency: Customer may asynchronously dispatch the `Address` before or during the delegation, and so this message will be incorrectly addressed to Agency. We call such messages "*lost messages*". Because of this lost message, Customer and Agency may have different views of the session being delegated at the time of delegation: performing reconnection naively breaks communication safety as Customer and Service have inconsistent session states. Therefore, a correct reconnection-based delegation protocol should be able to resolve potential session view discrepancies.

Two reconnection-based protocols. We have examined and implemented two strategies for resolving the issue of lost messages due to session view discrepancy.

Resending Protocol (resends cached lost messages after reconnection) Here lost messages, if any, are cached at and resent from the passive-party to the s-receiver after the new connection is established, explicitly re-synchronising session state before the delegated session is resumed. In our example, the original connection between Customer and Agency is first replaced by a connection between Customer and Service; then Customer resends the `Address` to Service and they resume the conversation.

Forwarding Protocol (forwards lost messages before reconnection) Here the s-sender first forwards all lost messages (if any) received from the passive-party, and then the delegated session is re-established. In our example, the `Address` is forwarded by Agency to Service and then the original connection is replaced by the new connection between Customer and Service.

4.2 Properties of the Delegation Protocols

The preceding discussion on lost messages suggests the value of being explicit about the correctness properties of the delegation protocols. We list the key properties against which the two designs are validated. Below "control message" means a message created by a delegation protocol, as opposed to the actual "application messages" of the conversation.

P1: Linearity For control message sent, there is always a unique receiver waiting for that message. Hence each control message is delivered deterministically without confusion.

P2: Liveness Discounting failure, the delegation protocol cannot deadlock, i.e.:
- (Deadlock-freedom) No circular dependencies between actions.
- (Readiness) The server side of the reconnection is always ready.
- (Stuck-freedom) The connection for the session being delegated is closed only after all messages have been completely resent or forwarded.

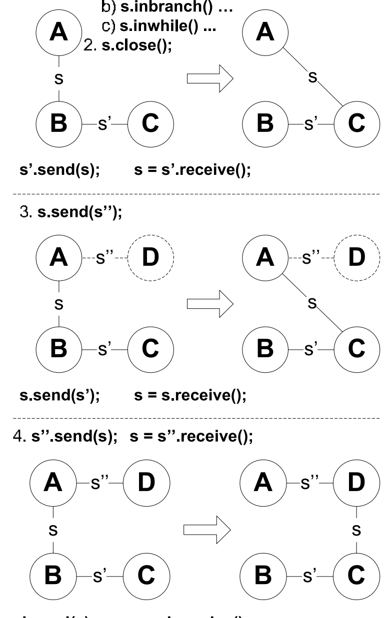
P3: Session Consistency The delegation protocol ensures no message is lost or duplicated, preserving the order and structure of the delegated session.

In the following discussions we assume that a protocol uses the same TCP connection for both application messages and control messages (as in our actual implementation). Semantically this means ordering is also preserved between both these kinds of messages.

4.3 General Framework for Reconnection-Based Delegation

To set up the general scenario, we let A be the passive-party, B the s-sender and C the s-receiver; B will delegate the session s to C via s'. The basic idea is for B to inform A that s is being delegated s via a *delegation signal* containing the address of C. Eventually the original connection for s is closed and A reconnects to the delegation target C. From this point, we shall simply say "A" to mean the "runtime for A" if no confusion arises.

Recalling that control and application messages are sequentialised, the delegation signal from B will only be detected by the runtime of A when blocked expecting some message from B as dictated by the session type. The subsequent behaviour of the delegation protocols depends on what the expected input should be. There are four cases, as illustrated below (the first picture corresponds to Cases 1 and 2, the second Case 3 and the third Case 4).



Case 1: A is performing a basic receive (for `receive`, including higher-order receive, `inbranch` or `inwhile`), waiting for a value, a session or a label.

Case 2: A has has finished its side of s. to be closed between A and C.

Case 3: A is attempting to delegate another session s'' to B via s, where s'' is with the fourth party D.

Case 4: A is also delegating the session s, to the fourth party D. This is *simultaneous delgeation.*

Case 3 comes from the fact that delegating a session is a compound operation that contains a blocking input.[3] Since A has to be waiting for an input to detect the delegation signal, we need not consider the cases where A is performing an atomic output operation (ordinary `send`, `outbranch` or `outwhile`).

As an example, we return to our example application: Customer is attempting to receive a `Date` (from Agency) when it detects the delegation signal, hence this is an instance of Case 1. Taking Case 1 as the default case for the purposes of this discussion, we shall illustrate the operation of the resending and forwarding protocols in § 4.4 and § 4.5. We then outline the remaining three cases in § 4.6.

4.4 Resending Protocol

The operation of the resending protocol for Case 1, as implemented given our existing design choices, is given in Figure 4. The key feature of this protocol is the use of session types at runtime to track the progress of the two peers of

[3] We have also implemented a non-blocking delegation. For simplicity, however, we consider only synchronous session-sending in the current discussions.

1. $\mathsf{B}{\to}\mathsf{C}$: "Start delegation"
2. C: open server socket on free port p_{C}, accept on p_{C}
3. $\mathsf{C}{\to}\mathsf{B}$: p_{C}
4. $\mathsf{B}{\to}\mathsf{A}$: $DS^{\mathsf{B}}_{\mathsf{A}}(\mathsf{C}) = \langle ST^{\mathsf{B}}_{\mathsf{A}}, IP_{\mathsf{C}}, p_{\mathsf{C}} \rangle$
5. $\mathsf{A}{\to}\mathsf{B}$: ACK_{AB}
6. A: close s | 6'. B: close s
7. A: connect to $IP_{\mathsf{C}} : p_{\mathsf{C}}$
8. $\mathsf{A}{\to}\mathsf{C}$: $LM(ST^{\mathsf{A}}_{\mathsf{B}} - ST^{\mathsf{B}}_{\mathsf{A}})$

Fig. 4. Operation of the Resending Protocol for Case 1.

the delegated session: this makes it possible to exactly identify the session view discrepancy ("the lost messages") and resynchronise the session.

The first phase of the protocol runs from Step 1 to Step 5, which delivers the information needed for reconnection and resending to the relevant parties. In Step 1, B informs C that delegation is happening. In Step 2, C binds a new `ServerSocket` to a fresh local port p_{C} for accepting the reconnection, and in Step 3, C tells B the value of p_{C}. In Step 4, B sends the delegation signal (for target C), denoted $DS^{\mathsf{B}}_{\mathsf{A}}(\mathsf{C})$, to A. As stated, this signal contains the runtime session type of the session being delegated, from B's perspective, denoted $ST^{\mathsf{B}}_{\mathsf{A}}$. As a result A can now calculate the difference between its view and B's view for this session. The delegation signal also contains the IP address and open port of the delegation target, IP_{C} and p_{C}. In Step 5, A sends an acknowledgement ACK_{AB} to B. This concludes the first phase.

The second phase performs the actual reconnection and lost message resending. Firstly, in Step 6 and Step 6', A (immediately after sending ACK_{AB}) and B (after receiving it) close their respective socket endpoints for the original session connection: any lost messages at B are simply be discarded. In Step 7, A connects to C to establish the new connection for the delegated session (C has been waiting for reconnection at p_{C} since Step 2). In Step 8, A resends the lost messages, denoted $LM(ST^{\mathsf{A}}_{\mathsf{B}} - ST^{\mathsf{B}}_{\mathsf{A}})$, to C based on the session type difference calculated above (after Step 4). A retrieves the lost messages from its cache of previously sent messages (maintained by each party), and C buffers them. In our running example, the runtime type $ST^{\mathsf{B}}_{\mathsf{A}}$ (the view from B) is T;`!{ACCEPT:!(Address)}`, and the runtime type $ST^{\mathsf{A}}_{\mathsf{B}}$ (the view from A) is T;`?{ACCEPT: }`. Hence the difference $ST^{\mathsf{B}}_{\mathsf{A}} - ST^{\mathsf{A}}_{\mathsf{B}}$ is `!(Address)`, and the corresponding message is resent after the reconnection. After Step 8, A and C can resume the session as normal.

4.5 Forwarding Protocol

In the forwarding protocol, A does not have to concern itself about lost messages as they are automatically forwarded from B (the old endpoint of the delegated session) to C (the new endpoint). The protocol works as listed in Figure 5.

The first phase of the protocol (Step 1 to Step 5) is precisely as in the resending protocol, except that the delegation signal in Step 4 no longer needs to carry the runtime session type $ST^{\mathsf{B}}_{\mathsf{A}}$.

In the second phase, reconnection is attempted in parallel with the lost message forwarding. In Step 5', which immediately follows Step 4 (sending the del-

1. $\mathsf{B}\!\to\!\mathsf{C}$: “Start delegation”					
2. C: *open server socket on free port* p_{C}					
3. $\mathsf{C}\!\to\!\mathsf{B}$: p_{C}					
4. $\mathsf{B}\!\to\!\mathsf{A}$: $DS^{\mathsf{B}}_{\mathsf{A}}(\mathsf{C}) = \langle IP_{\mathsf{C}}, p_{\mathsf{C}} \rangle$					
5. $\mathsf{A}\!\to\!\mathsf{B}$: ACK_{AB}	5’.	B: enter f/w mode			
6. A: close s	6’.	$\mathsf{B}\!\to\!\mathsf{C}$: $\tilde{V}{::}ACK_{\mathsf{AB}}$			
7. A: connect to $IP_{\mathsf{C}}\!:\!p_{\mathsf{C}}$	7’.	B: exit f/w mode	7”.	C: buffer $\tilde{V}$	
	8’.	B: close s	8”.	C: accept on p_{C}	

Fig. 5. Operation of the Forwarding Protocol for Case 1.

egation signal to A), B starts forwarding to C all messages that have arrived or are newly arriving from A. The actual delivery is described in Step 6’ where $\tilde{V}$ denotes all messages received by B from A up to ACK_{AB}, i.e. the “lost messages”. The delegation acknowledgment ACK_{AB} sent by A in Step 5 signifies end-of-forwarding when it is received and forwarded by B to C in Step 6’: B knows that A is aware of the delegation and will not send any more messages (to B), and hence ends forwarding in Step 7’. $\tilde{V}$ is buffered by C to be used when the delegated session is resumed.

In Step 6, A closes its endpoint to the connection with B after sending ACK_{AB} in Step 5; since B may still be performing the forwarding at this point, the opposing endpoint is not closed until Step 8’. In Step 7, A requests the new connection to C using the address and port number received in Step 4. However, C does not accept the reconnection until Step 8” (p_{C} is open so A blocks) after receiving all the forwarded messages in Step 7”. As for the resending protocol, after the session is resumed C first processes the buffered messages $\tilde{V}$ before any new messages from A, preserving message order. Note Steps 5-7, Steps 5’-8’ (after Step 4) and Steps 7”-8” (after Step 6’) can run in parallel with two cross-dependencies, 6’ on 5 and 8” on 7.

4.6 The Remaining Cases for the Delegation Protocols

We summarise how the two protocols behave for the remaining three cases discussed in § 4.3. The full protocol specifications can be found in [21]. Most parts of both protocols are identical with Case 1: the key idea is that the role of the delegation acknowledgement ACK_{AB} is played in each case by some other control signal.

In *Case 2*, A sends a special signal FIN_{AB} (due to the close protocol) to let B know that it has completed its side of the session. Basically FIN_{AB} signifies instead of ACK_{AB} to B that the original session connection can be closed immediately (hence Step 5 is not needed).

In *Case 3*, A is the s-sender for another session s'' between A and the fourth party D (the passive-party of s''). In this case, B receives a “Start Delegation” signal (for the delegation of s'') from A. In the resending protocol, this signal is resent with $LM(ST^{\mathsf{A}}_{\mathsf{B}} - ST^{\mathsf{B}}_{\mathsf{A}})$ at Step 8 to C in order to start the subsequent run of the delegation protocol with A and D. In the forwarding protocol, this message simply replaces ACK_{AB} as an end-of-forwarding signal after being forwarded by B, and at the same time alerts C to the subsequent delegation.

In *Case 4*, instead of $ACK_{\rm AB}$ at Step 5, B receives $DS^{\sf A}_{\sf B}({\sf D})$ from A. In the resending protocol, C buffers the lost messages from A, closes this intermediate connection, and then reconnects to the port at which D is waiting (C gets the address of D from A). The forwarding protocol is similar to *Case 3*.

4.7 Correctness of the Delegation Protocols.

We present the key arguments for our protocols w.r.t. the three properties **P1-3** in § 4.2, focusing on *Case 1*. For other cases see [21].[4]

Resending Protocol: P1 is obvious from the description of the protocol. For **P2** we first observe concurrent executions only exist between Step 6-8 and Step 6'. Note a deadlock arises only when a cycle (like A→B and B→A) is executed concurrently which is impossible from the protocol definition. Readiness holds since the connection to $p_{\sf C}$ (Step 7) takes place after its creation (as Step 2). Stuck-freedom holds since Steps 6 and 6' take place after all operations are completed between A and B, ensured by $ACK_{\rm AB}$. The key property is **P3**. This holds since the sending action from C happens after the lost messages from Step 8 are stored at C, which holds since the sending action from C uses the same port $p_{\rm C}$. Hence the order of the session is preserved before and after the protocol.

Forwarding Protocol: The reasoning is essentially similar to that for the resending protocol, especially for **P2**. The key property is now **P1** which further affects **P3**. Basically, observe that positioning "accept $p_{\rm C}$" after all forwarding has been completed means that there cannot be any communication on the new connection to interfere with message ordering in the session (i.e. in Step 8"), we avoid this preposterous situation, satisfying both **P1** and **P3**. If we had "accept $p_{\rm C}$" before Step 8" so that the next value from C to A via $p_{\rm C}$ were to be sent without waiting for the ack from Step 6', C could receive further messages from A ahead of the forwarded message at Step 7, destroying **P1**. This would have confused the ordering of the session, making **P3** unsatisfied.

4.8 Trade-Offs in the Two Protocols

A central contrast between the two protocols lies in which party takes responsibility for recovering lost messages: in the resending protocol the responsibility is on A (passive-party) while in the forwarding protocol it lies on B (s-sender, the old endpoint). We examine how this issue affects trade-offs between cost and robustness in the two protocols, suggesting different environments and requirements these protocols may be better suited to.

Regarding space-time cost, the resending protocol requires runtime session type tracking and caching previously sent messages. However, by session types, we need only cache up to the most recent receive operation, and the cost can be predicted. On the other hand, if there is a chain of the delegations, the forwarding protocol requires $\tilde{V}$ to go through all peers. Both protocols use the same number of control messages in total.

[4] N.B. Deadlock can occur if, in Cases 1-4, A and C are the same party, or if, in Case 4, C and D are the same party. We discount these cases; the delegation of a session to the peer of that session is illegal [18] and gives a runtime error.

Regarding robustness, the resending protocol allows B to exit at an earlier stage, which can be an advantage in mobile computing (e.g. if B needs to move to a different location). Furthermore, the s-sender performs more operations with the s-receiver in the forwarding protocol which means the failure of one may affect the other as well as the passive-party, which is clearly less desirable.

5 Performance and Experience

The current implementation of SJ incorporates all of the compiler and runtime elements discussed so far including the two delegation protocols, resending and forwarding (called `SJRSocket` and `SJFSocket`). This section evaluates the session runtime performance through benchmarks in low and high latency environments. We focus on one of the benchmarks which compares simple SJ-programs with equivalent programs implemented in Java Socket (`java.net.socket`) and Java RMI. The results for FSocket show that the session abstraction can be efficiently supported over TCP; RSocket, in spite of its additional runtime overhead and lack of optimisation, is consistently competitive with RMI.

Protocol and benchmark plan. For our main micro benchmark we used a simple protocol of type `begin.![ ?(MyObject) ]*`, which approximately corresponds to an RMI method of signature, `MyObject rmiMeth(boolean b)`. The primitive boolean argument of the method is implicit in the session type `![...]*`, and hence the communication of this message is common to both session and RMI. By a *session of length* n we mean n iterations of the in/outwhile loop, or n consecutive RMI calls. The SJ and standard socket also incur one extra communication for the final `false` value that ends the conjoined iteration, whereas the RMI client simply stops calling the server. The full source code for the benchmark are available at [21].

We measured the time to complete sessions of increasing length (starting at zero, and increasing orders of magnitude) for different sizes of `MyObject` (starting from a serialised size of 100 Bytes: for reference, a `java.lang.Integer` object serialises to 81 Bytes). For each combination of the session length and message size, the benchmark was executed $1,000$ times and the mean of the results, excluding the first run (for stability), were recorded. Each run of a session is performed by a separate instance of the client program, ensuring that a new connection is used each time; the server sides operate continuously in all the cases. Each session is preceded by a dummy run of length one (a single dummy call in the RMI case) to ensure all required classes have been loaded and verified by the JVM at both the server and client before executing the actual benchmarked session. The communicating parties were set up so that no class downloading is required.

The same benchmarks were performed in both low latency and high latency environments. In the former the benchmarks were executed on two physically neighbouring PCs (Intel Pentium 4 HT 3 GHz, 1 GB main memory) connected via gigabit Ethernet, running Mandrake 10.2 (Linux 2.6.17) with Java compiler and runtime (Standard Edition) version 1.6.0. Latency between the two machines was measured using ping (56 bytes) to be on average 0.14 ms. Nagle's algorithm is

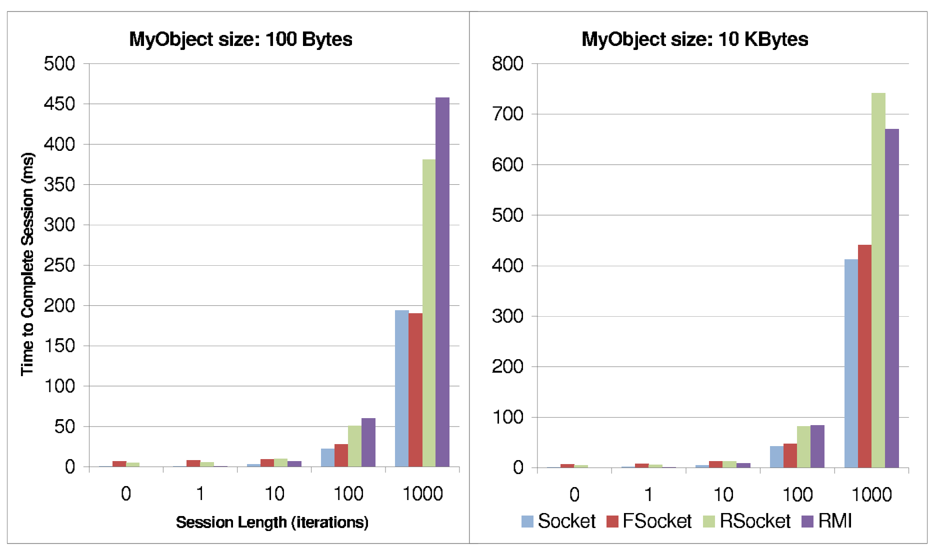


Fig. 6. Benchmark results for message sizes 100 Bytes and 10 KBytes.

disabled (`TCP_NODELAY` is set to true) in SJ and raw socket. All of our benchmarks avoid DNS lookups by directly using IP addresses.

The benchmark results for `MyObject` of sizes 100 Bytes and 10 KBytes are given in Figure 6; the full data set is available at [21]. The forwarding-based session runtime (FSocket) incurs very little overhead against the standard socket implementation (Socket). RSocket is more costly than FSocket, as the RSocket runtime additionally tracks the types of active sessions and manages the cache for copies of previously sent messages (however the number of cached messages never has to exceed one as the cache is cleared with each iteration using runtime type information). This process currently involves an extra serialisation step which can be avoided. The overheads of RSocket are slightly more prominent for smaller `MyObject` sizes; this is because the relative cost of the actual communication is less. Note that RSocket, although quite unoptimised, is still consistently competitive with RMI. We also observe that many applications will naturally involve greater local processing so that the relative overhead of communication in SJ will be offset through interleaving.

The results from running the same benchmark in a higher latency environment (RTT ∼30ms), available at [21], reinforce the above findings: a higher latency increases the cost of basic communication and so offsets (substantially eliminates) the relative cost of the session runtime overheads. Further the results on FSocket in comparison to bare Socket show that the overheads for our current implementation of session initiation based on asynchronous handshake are practically negligible for a larger session (of length 100 and more). Further micro benchmark results including those on delegation performance (consistent with our architectural design discussed in Section 4) are presented in [21].

Programming experience. In addition to Travel Agency in § 2, we used SJ to implement the major usecases from [5], all of which are sophisticated forms of business protocols. For some of these, we also implemented the equivalent versions using standard Socket (each of these examples involves a structured sequence of asynchronous messages, as is standard in many financial protocols [31], so that coding them with RMI is not as natural). These examples include a more complex version of the Travel Agency example in [5], which features branching and nested iteration with two possible break points from the different loop levels. Another usecase has more concurrent features, where one buyer interacts with multiple suppliers who in turn interact with multiple manufactures.

From these examples, we find three key aspects which make session-based programming considerably easier than bare socket programming.

1. *The direct use of classes as the types of messages.*
2. *The high-level abstraction of session operations (in particular delegation), mapped to transport communication actions by the session runtime.*
3. *The explicit declaration of the conversation structure and accompanying static validation.*

All these aspects contribute to a simpler, pain-free and streamlined programming of communications. In particular we find many conversation patterns (including a large real usecase as found in [15, Chapter 7]) are described clearly and concisely by a combination of sequencing, branching, recursive sessions and simple exceptions with existing Java primitives, ensuring type-safety. See [21] for the full sample programs.

Our current implementations are not optimised in such aspects as the global set-up time, but our micro benchmark results suggest that SJ has the prospect for the use in real business environments with competitive performance, with clear and succinct description of communication structures, robust type checking, and potential enhancement in resilience and other properties through session runtime features such as reconnection-based delegation.

6 Related Work

Language design for session types One of the usage of session types in practice is found in Web Services. Because of the need of static validation of safety of business protocols, a description language called Web Service Description Languages (WS-CDL), developed by a W3C standardisation working group [28], uses a variant of session types. A description in WS-CDL is implemented through communication among distributed end-points written in languages such as Java or WS4-BPEL. [8, 6, 19] studied the principles to obtain a sound and complete implementation from a description written in CDL. Another use of session-types is the standardisation of financial protocols in UNIFI (ISO20022) [31]. We are planning to use SJ and its compiler-runtime as a part of implementation technologies for these standards.

An implementation of session types in Haskell is studied in [24], where a calculus based on session types is encoded into Haskell. A merit of this approach

is that a type checking for session types can be done by that for Haskell. They do not consider compatibility check at session initiation, which is essential for using session types in open environments. It may be difficult to realise compatibility check or type-safe delegation within their framework since their encoding does not directly type IO channels.

Fähndrich et. al [14] integrate a variant of session types into a derivative of C♯ for systems programming in shared memory uni/multiprocessor environments, with an aim to describe interface among OS-modules as message passing conversations. The focus of their design is to implement message passing in shared memory uni/multiprocessors rather than in distributed computing environments. To realise their aim, they use fixed conversation structures given by session types in combination with ownership types and a designated heap area for messages: in-session communications are executed as direct pointer rewriting, obtaining efficiency suitable for shared memory kernel programming. From the viewpoint of abstraction for distributed object-oriented programming, their design lacks dynamic type checking and subtyping as found in SJ, both of which are essential in open distributed environments. Since session-based communication in [14] assumes shared memory environments, its runtime does not include distributed implementation of session abstractions such as protocols for delegation, which is a focal point of the present study. In spite of significant differences in intended environments and design directions, the two works show non-trivial impacts the introduction of session types can have on abstraction and implementation in objected-oriented languages.

A framework of cryptographically protecting session execution from both external attackers in networks and malicious principals is studied based on F# in [12]. Their session specification models an interaction sequence between two or more constituent network peers (called *roles*). The description is given as a graph whose node represents a state of a role in a session, and whose edge denotes a dyadic communication and control flow. Their aim is to use such specifications for modelling and validation rather than programming. Thus neither a type discipline nor a runtime are implemented in their work.

Language design based on process calculi The present work shares with many recent works its direction towards well-structured communication-based programming using types. Pict [25] is the programming language based on the π-calculus with linear and polymorphic types. Polyphonic C♯ [7] is designed based on Join-calculus and uses a type discipline for safe and sophisticated object synchronisation. Acute [1] is an extension of OCaml for coherent naming and type-safe version change of distributed code. Concurrency and Coordination Runtime (CCR) [2] is a port-based concurrency library for C♯ for component-based programming, whose design is based on Poly♯.

Occam-pi [3] is a highly efficient concurrent language based on channel-based communication. Its syntax is based on both Hoare's CSP (and its practical embodiment, Occam) and the π-calculus. Designed for systems-level programming, the language allows generation of more than million threads for a single processor machine without efficiency degradation. Occam-pi can realise various locking

and barrier abstractions built from its highly efficient communication primitives. DirectFlow [23] is a domain specific language which supports stream processing with a set of abstractions inspired by CSP, such as filters, pipes, channels and duplications. DirectFlow is not a stand-alone programming language, but is used via embedding into any host languages for defining a data-flow of components. In both languages, typing of a larger unit of (a series of) hand-shake communications than an individual communication or composition has not been guaranteed.

X10 [10] is a typed concurrent language based on Java, and is designed for high-performance computing. Its current design focuses on global, distributed memory whose sharing is carefully controlled by X10's type system. A notable aspect is the introduction of distributed locations into the language, cleanly integrated with its disciplined thread model. The current version of the language does not include communication primitives.

None of the above works use conversation-based abstraction for communication programming, hence neither typing disciplines that can guarantee communication safety of a conversation structure, nor associated runtime for realising the abstraction is considered. The interplay between their design elements and session types is an interesting future topic.

7 Conclusion and Future Work

This paper presented the design and implementation of session types and associated programming constructs in Java. Session types can be used to declare a variety of structured interactions between multiple distributed parties and naturally integrated with object-oriented distributed programming such as class-downloading, with compositional, static and dynamic type-checking. We extended Java with session primitives and with subtyping. Our experience so far indicates that many programming patterns for structured conversations [6, 4, 31, 15] are representable by a combination of branching, iteration and basic exceptions, together with session subtyping. We also designed and implemented two alternative session delegation protocols with correctness arguments. Lastly, we demonstrated that our runtime framework can support the session abstraction with minimal overhead on the underlying transports, and our current implementation performs competitively with RMI, the standard typed primitives for inter-process communications in object-oriented languages.

Sessions and session types offer an abstraction layer for structured communication sequences. Our runtime decouples underlying communication mechanisms from this layer, mediated by type information. Exploitation of this framework for contexts other than socket is an interesting subject of further study, especially given the inherently open nature of communication. Safety of communication in a session hinges on two assumptions: that involved programs are statically verified with respect to declared session types, and that they communicate their session types and perform the compatibility validation honestly. Either maliciously or by error, if these assumptions are not met, safety is lost. How we can dynamically avoid this issue is an interesting future topic, together with the study of other aspects of security in session types [12]. Investigation of how session-based

programming may be integrated into such languages as [10, 3, 23, 29] is also an open subject. A logical nature of sessions, used for transparent delegation in the present work, can be exploited along the line of [27], making the most of the predictability of interactions based on session types. This logical nature can in turn be used for optimisations: In addition to basic code refactoring, advantage of session type information regarding types (hence often size bound) and direction of messages as well as predetermined scenarios of communications may be exploited. For example, we can piggyback the first message(s) of a session on the initiation messages, to compensate for the initiation overheads, which will be effective in low-latency environments. Object serialisation might be customised to generate smaller binary data with less embedded type information given that the expected type of the message is known a priori. There are many other opportunities for optimisations including message batching (like Nagle's algorithm, but based on session types), which may give better performance for small messages in high latency environments. The current implementation and its refinement are being developed as a possible foundation of programming and execution for public standards for web services [6] and financial protocols [31], combined with theories from [8, 20].

References

1. Acute home page. `http://www.cl.cam.ac.uk/users/pes20/acute`.
2. CCR: An Asynchronous Messaging Library for C#2.0. `http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime`.
3. occam-pi home page. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.
4. Web Services Choreography Description Language: Primer 1.0. `http://www.w3.org/TR/ws-cdl-10-primer/`.
5. Web Services Choreography Requirements. `http://www.w3.org/TR/ws-chor-reqs/`.
6. Web Services Choreography Working Group. `http://www.w3.org/2002/ws/chor/`.
7. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
8. M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, volume 4421 of *LNCS*, pages 2–17. Springer, 2007.
9. M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. Published in [6], 2006.
10. P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA'05*. ACM Press, 2005.
11. M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. In *FMOODS'07*, volume 4468 of *LNCS*, pages 1–31, 2007.
12. R. Corin, P.-M. Denielou, C. Fournet, K. Bhargavan, and J. Leifer. Secure Implementations for Typed Session Abstractions. In *CFS'07*. IEEE-CS Press, 2007.
13. M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer, 2006.

14. M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. C. Hunt, J. R. Larus, , and S. Levi. Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys'06*, ACM SIGOPS, pages 177–190, 2006.
15. H. Foster. *A Rigorous Approach to Engineering Web Service Compositions*. PhD thesis, Department of Computing, Imperial College London, January 2006.
16. S. Gay and M. Hole. Types and Subtypes for Client-Server Interactions. In *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.
17. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
18. K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.
19. K. Honda, N. Yoshida, and M. Carbone. Web Services, Mobile Processes and Types. *The Bulletin of the European Association for Theoretical Computer Science*, February(91):165–185, 2007.
20. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types, 2008. *POPL'08*, `http://www.doc.ic.ac.uk/~yoshida/multiparty.html`.
21. R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. `http://www.doc.ic.ac.uk/~rh105/sessiondj.html`.
22. IETF. IP Mobility Support for IPv4. `http://dret.net/rfc-index/reference/RFC3344`.
23. C.-K. Lin and A. P. Black. DirectFlow: A domain-specific language for information-flow systems. In *ECOOP*, volume 4609 of *LNCS*, pages 299–322. Springer, 2007.
24. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
25. B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*. MIT Press, 2000.
26. Polyglot Home Page. `http://www.cs.cornell.edu/Projects/polyglot/`.
27. A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, pages 155–166, 2000.
28. S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), March 2006.
29. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in java. In *OOPSLA*, pages 211–228. ACM, 2007.
30. K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In *PARLE'94*, volume 817 of *LNCS*, pages 398–413, 1994.
31. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.