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 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 and interleaving, combined with class downloading and failure handling. The compilation-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 mechanisms for performing distributed session delegation and prove their correctness. Benchmark results show session abstraction can be realised with low runtime overhead.

1 Introduction

Communication in object-oriented programming. Communication is becoming a fundamental element of software development. Web applications increasingly combine numerous distributed services; an off-the-shelf CPU will soon host hundreds of cores per chip; corporate integration builds complex systems that communicate using standardised business protocols; and sensor networks will place a large number of processing units per square meter. A frequent pattern in communication-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 communications, 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 [12, 13, 27] 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; a session type is a specification of the structure and message types of a 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 into object-oriented programming in Java. Preceding works include theoretical studies of session types in object-oriented core calculi [8, 10], and the implementation of a systems-level object-oriented language with session types for shared memory concurrency [11]. 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 and interleaving, combined with class downloading and failure handling. The following summarises the central features of the proposed compilation-runtime framework.

1. *Integration of object-oriented and session programming disciplines.* We extend Java with concise and clear syntax for session types and structured communication operations. Session-based distributed programming involves specifying the intended interaction protocols using session types and implementing these protocols using the session operations. The session implementations are then verified against the protocol specifications. This methodology uses session types to describe interfaces for conversation in the way Java interfaces describe interfaces for method-call interaction.
2. *Ensuring communication safety for distributed applications.* Communication safety is guaranteed through a combination of static and dynamic validations. Static validation ensures that each session implementation conforms to a locally declared protocol specification; runtime validation at session initiation checks the communicating parties implement compatible protocols.
3. *Supporting session abstraction over concrete transports.* Our compilation-runtime framework 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 session abstraction can be realised over the underlying transport with low runtime overhead.

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 session abstraction above the underlying transport. We have designed and implemented two alternative mechanisms for performing delegation, and proved their correctness. We also demonstrate how the integration of session types and objects can support extended features such as eager remote class loading and eager class verification.

Paper summary. Section 2 illustrates the key features of session programming by example. Section 3 describes the design elements of our compilation-runtime framework. Section 4 discusses the implementation of session delegation and its correctness. Section 5 presents benchmark results. Section 6 discusses related work, and Section 7 concludes. The compiler and runtime, example applications and omitted details are available at [26].

2 Session-Based Programming

This section illustrates the central ideas of programming in our session-based extension of Java, called 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 [6], capturing a collaboration pattern typical to many business protocols [3, 28]. 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 through the use of *session delegation.* Delegation in SJ enables dynamic mobility of sessions whilst preserving communication safety. The overall scenario of this conversation is 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 initiative of Customer.
2. Customer either accepts an offer from Agency or decides that none of the received quotes are satisfactory (these 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* to Service, through s', the interactions with Customer remaining for s. The particular travel service contacted by Agency 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 programmed in SJ. Roughly speaking, session programming consists of two steps: specifying the intended interaction protocols using session types, and implementing these protocols using session operations.

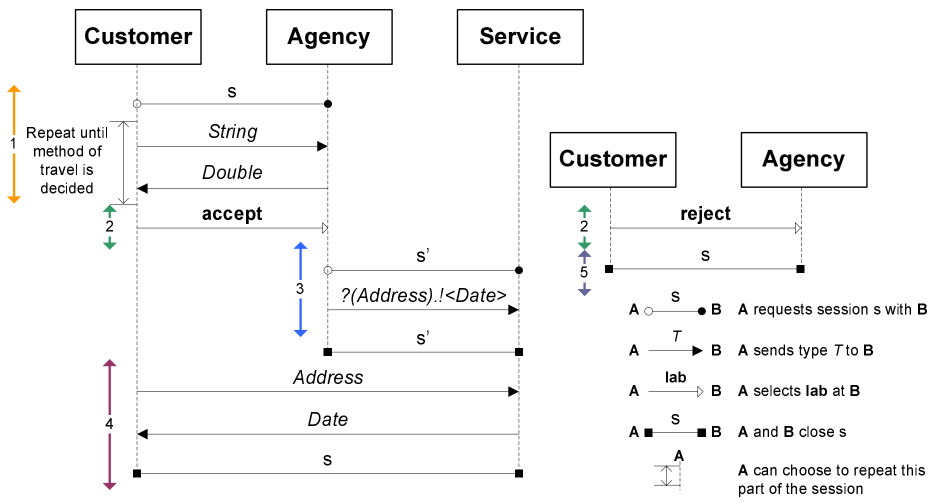
Customer
Agency
Service
s
Repeat until method of travel is decided
1
String
Double
accept
2
3
s'
?(Address).!<Date>
s'
4
Address
Date
s
Customer
Agency
reject
2
5
s
A s B A requests session s with B
A T B A sends type T to B
A lab B A selects lab at B
A s B A and B close s
A
A can choose to repeat this part of the session

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

Protocol specification. In SJ, session types are called *protocols*, which are declared using the `protocol` keyword. The protocols for the order session (between Customer and Agency) are specified below as `placeOrder`, which describes the interactions from Customer's side, and `acceptOrder`, from Agency.[3]

```
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.

We first look at `placeOrder`: the first part says Customer can repeat as many times as desired (expressed by `![..]*`), the sequence 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. The `acceptOrder` protocol is *dual* to `placeOrder`, given by inverting the input '`?`' and the output '`!`' symbols in `placeOrder`, thus guaranteeing a precise correspondence between the actions of each protocol.

[3] SJ also supports an alternative syntax for protocols (session types) that replaces the symbols such as '`!`' and '`?`' with keywords in English [26].

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

- *Session server socket* of class `SJServerSocket`, which listens for session requests, accepting those compatible with the specified protocol.
- *Session server-address* of class `SJServerAddress`, which specifies the address of a session server socket and the type of session it accepts; and
- *Session socket* of class `SJSocket`, which represents one endpoint of a session channel, through which communication actions within a session are performed. Clients use session sockets to request sessions with a server.

SJ uses the terminology from standard socket programming for familiarity. The *session sockets* and *session server sockets* correspond to their standard socket equivalents, but are enhanced by their associated session types. Client sockets are bound to a session server-address at creation, and can only make requests to that server. Session server sockets accept a request if the type of the server is compatible with the requesting client; the server will then create a fresh session socket (the opposing endpoint to the client socket) for the new session. Once the session is established, messages sent through one socket will be received at the opposing endpoint. Static type checking ensures that the sent messages respect the type of the session; together with the server validation, this guarantees communication safety. The occurrences of a session socket in a SJ program clearly delineate the flow of a conversation, interleaved with 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);
```

After opening a server socket, the server party can accept a session request by,

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

where `s_ac` is an uninitialised (or null) SJSocket variable. The `accept` operation blocks until a session request is received: the server then validates that the protocol requested by the client is compatible with that offered by the server (see § 3 for details) and returns a new session socket, i.e. the server-side endpoint.

Session server-address and session sockets. A session server-address in the current SJ implementation identifies a server by its IP address and TCP port. At the Customer, we set:

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

A server-address is typed with the session type seen from the client side, in this case `placeOrder`. Server-addresses can be communicated to other parties, allowing them to request sessions with the same server. Customer uses `c_ca` to create a session socket:

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

and request a session with Agency:

```
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 exchange session types, independently validate compatibility, and if successful the session between Customer and Agency is established. If incompatible, an exception is raised at both parties (see 'Session failure' below).

Session communication (1): send and receive. After the session has been successfully 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 protocol `placeOrder`. Static session type checking ensures that this contract is obeyed modulo session subtyping (see § 3.2 later).

The basic message passing operations, performed by `send` and `receive`, asynchronously communicate typed objects. 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)
```

In this instance, the compiler infers the expected receive type from the `placeOrder` protocol; explicit receive-casts are also permitted.

Session communication (2): iteration. Iteration is abstracted by the two mutually dual types written `![...]*` and `?[...]*` [8, 10]. Like regular expressions, `[...]*` expresses that the interactions in `[...]` may be iterated zero or more times; the `!` prefix indicates that this party controls the iteration, while its dual party, of type `?[...]*`, follows this decision. These types are implemented using the `outwhile` and `inwhile` [8, 10] operations, which can together be considered a distributed version of the standard while-loop. The opening exchange of `placeOrder`, `![!<String>.?(Double)]*`, and its dual type at Agency can be implemented as follows.

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

```
s_ac.inwhile() {
  String journeyDetails
      = 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 case, from Customer to Agency), synchronising the control flow between two parties.

Agency is programmed with the dual behaviour: `inwhile` does not specify a loop-condition because this decision is made by Customer and communicated to Agency at each iteration. These explicit constructs for iterative interaction can greatly improve code readability and eliminate subtle errors, in comparison to ad hoc synchronisation over untyped I/O.

Session communications (3): branching. A session may branch down one of multiple conversation paths into a sub-conversation. In `placeOrder`, Customer's type reads `!{ACCEPT: !<Address>.?(Date), REJECT: }`, where `!` signifies the selecting side. Hence, Customer can select `ACCEPT`, proceeding into a sub-conversation with two communications (send an Address, receive a Date); otherwise, selecting `REJECT` immediately terminates the session.

The branch types are implemented using `outbranch` and `inbranch`. This pair of operations can be considered a distributed switch-statement, or one may view `outbranch` as being similar to method invocation, with `inbranch` likened to an object waiting with one or more methods. We illustrate these constructs through 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:

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

The session-try refines the standard Java try-statement to ensure that multiple sessions, which may freely interleave, are consistently completed within the specified scope. Sessions may fail at initiation due to incompatibility, and later at any point due to I/O or other errors. Failure is signalled by propagating terminal exceptions: the failure of one session, or any other exception that causes the flow of control to leave the session-try block, will cause the failure of all other ongoing sessions within the same scope. This does not affect a party that has successfully completed its side of a session, which will asynchronously leave its session-try scope. Nested session-try statements offer programmers the choice to fail the outer session if the inner session fails, or to consume the inner exception and continue normally.

Session delegation. If Customer is happy with one of Agency's quotes, it will select `ACCEPT`. This causes Agency to open a second session with Service, over which Agency delegates to Service the remainder of the conversation with Customer, as specified in `acceptOffer`. After the delegation, Agency relinquishes the session and Service will complete it. Since this ensures that the contract of the original order session will be fulfilled, Agency need not perform any further action for the delegated session; indeed, Agency's work is finished after delegation.

At the application-level, the delegation is exposed to 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 the following mutually dual protocols:

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

Delegation is abstracted by *higher-order session types* [8, 10, 13], where the specified message type is itself a session type; in this example, `?(Address).!<Date>`. The message type denotes the unfinished part of the protocol of the session being delegated; the party that receives the session will resume the conversation, according to this partial protocol.

In terms of implementation, delegation is naturally represented by sending 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 = ... // delegateOrderSession.
  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(..., port)
SJSocket s_sa = ss_sa.accept();
SJSocket s_sc = s_sa.receive(); // Receive the delegated session.
```

Service then completes the session with Customer.

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

The SJ runtime incorporates two alternative mechanisms for delegation: § 4 discusses in detail the protocols by which these mechanisms coordinate the session parties involved a delegation.

The example covered in this section illustrates only the basic features of SJ. The full source code, compiler and runtime are available at [26], as well as further SJ programs featuring more complex interactions, including the implementation of business protocols from [6].

3 Compiler and Runtime Architecture

3.1 General Framework

The compilation-runtime framework of SJ works across the following three layers, in each of which session type information plays a crucial role.

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

By going through these layers, 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 the communication primitives that are supported by the session runtime interface. Layer 3 implements the session runtime interface over concrete transports and performs dynamic session typing. The compiler comprises approximately 15 KLOC extension to the base Polyglot framework [21]. The current implementation of the session runtime over TCP consists of approximately 3 KLOC of Java.

A core principle of this framework is that explicit declaration of conversation structures, coupled with the static and dynamic type-based validations, provides a basis for well-structured communication programming. We envisage programmers working on a distributed application first agree on the protocols, specified as session types, through which the components interact, and against which the implementation of each component is statically validated. Dynamic type checking, performed at runtime when a session is initiated, then ensures that the components are correctly composed: session peers must implement compatible protocols in order to conduct a conversation. The mechanisms encapsulated by the session runtime, which realise the major session operations (initiation, send/receive, branch, loop, delegation) as well as additional features (eager class downloading, eager class verification), 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. Then using this type information, it maps the SJ surface syntax to Java and the session runtime APIs. Type-checking

```
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.

for session 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 verifies that each session implementation conforms to the specified protocol with respect to the message types and conversation structure, according to the correspondence between session operations and type constructors given in Figure 2. As we discuss below, protocol declarations include all the information required for static type-checking of higher-order session communication (delegation).

We illustrate some of the notable aspects of static session verification. Firstly, when session sockets are passed via session delegation and as arguments to a method call (the latter is an example of integrating session and object types), the typing needs to guarantee a correct transfer of responsibility for completing the remainder of the session being passed. For this purpose, a method that accepts a session socket argument simply specifies the expected session type in place of the usual parameter type, e.g.

```
void foobar(!<int>.?(String) s, ...) throws SJIOException {
  ... // Implementation of ‘s’ according to !<int>.?(String).
}
```

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. Similarly the following examples are ill-typed since they delegate the outer session s2 multiple times.

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

However, delegation of nested sessions within an iteration is typable.

The type checking in SJ fully incorporates *session subtyping*, which is an important feature for practical programming, permitting message type variance for send and receive operations as well as structural subtyping for branching and selection [4, 12]. Message type variance follows the object class hierarchy, integrating object types into session typing; intuitively, a subtype can be sent where a supertype is expected. Structural session subtyping [3, 12] has two main purposes. Firstly, an outbranch implementation only selects from a subset of the cases offered by the protocol, and dually 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 these two kinds of session subtyping.

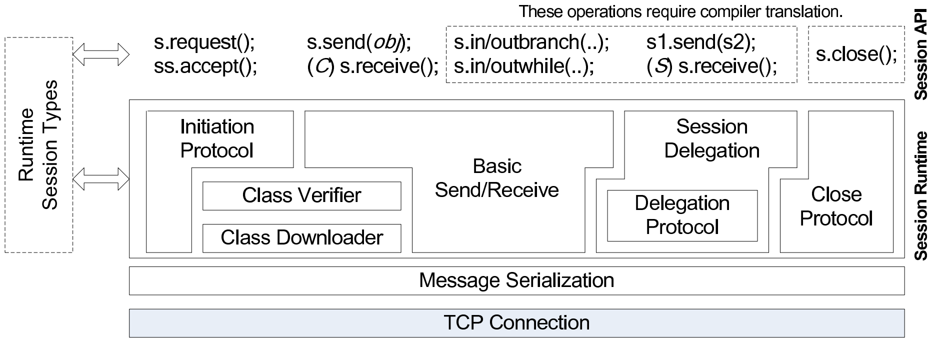

Fig. 3. The structure of the SJ session runtime.

```
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: { ... }
}
```

This example demands the use of subtyping at both compilation and runtime. Message subtyping is augmented by remote class loading, discussed in § 3.3.

3.3 The Session Runtime

The structure of the session runtime is illustrated in Figure 3. The SJ compiler translates the session operations, depicted as 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 together model the semantics of the translated operations: these protocols effectively 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 of the session runtime include:

- *basic send/receive*, which also supports in/outbranch and in/outwhile;
- the *initiation protocol*, *class verifier* and *class downloader*;
- 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 for delegation which is discussed in detail 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 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. Basic send/receive is also asynchronous, meaning a basic send does not block on the corresponding receive; however, receive does block until a (complete) message has been received. As depicted in Figure 3, our current implementation uses TCP: each active session is supported by a single underlying TCP connection. Session messages, either Java objects or primitive types, are transmitted through the TCP byte stream using standard Java serialization. Logically, we can use other transports; little modification would be required for transports that also support the above properties, such as SCTP.

Session initiation and dynamic type checking. Session initiation takes place when the `accept` and `request` operations of a server and client interact. The initiation protocol establishes the underlying TCP connection and then verifies session compatibility. The two parties exchange the session types which they implement and which have already been statically verified, and each independently validates compatibility modulo session subtyping. If successful, the session is established, otherwise both parties raise an exception and the session is aborted. The runtime also supports monitoring of received messages against the expected type. The initiation protocol can 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 a 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 choose to *eagerly* load, at session initiation, all potentially 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 choose to perform eager class verification for all shared classes at session initiation; class verification is currently implemented using the standard `SerialVersionUID` checks for serializable classes.

The close protocol. SJ does not have an explicit session close operation in its surface syntax; instead, a session is implicitly closed when a session terminates.

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 in an enclosing session-try scope, signalling session failure. In this case, the close protocol is responsible for propagating a failure signal 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. § 4 discusses how the delegation and close protocols interact in this case.

4 Protocols for Session Delegation

Session delegation is a defining feature of session-based programming; transparent, type-safe endpoint mobility raises session abstraction above ordinary communication over a concrete transport. This means that a conversation should continue seamlessly regardless of how many times session peers are interchanged at any point of the conversation. Consequently, each session delegation involves intricate coordination among three parties, or even four if both peers simultaneously delegate the same session. This section examines implementation strategies for delegation, focusing on the protocols for two alternative mechanisms that realise the above requirements, and presents key arguments for their correctness.

In the rest of this section, we use the following terminology. The *s-sender* stands for session-sender; the *s-receiver* for session-receiver; and the *passive-party* for the peer of the s-sender in the session being delegated. Recall that delegation is transparent to the passive party. Our design discussions assume a TCP-like connection-based transport, in accordance with the communication characteristics of session channels: asynchronous, reliable and order-preserving.

4.1 Design Strategies for Delegation Mechanisms

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. This is similar to Mobile IP [16]. The merit of this approach is that no extra actions are required on the part of the runtime 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 a failure of the s-sender also fails the delegated session.

An alternative strategy is to reconnect the underlying transport connections so that they directly 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). This mechanism demands additional network operations on the part of the runtime of the passive-party: at the same time, it frees the s-sender from the obligation to indefinitely take care of the delegated session. Reconnection precludes (long-running) rerouting of messages, which can have significant cost if many message exchanges are expected after the delegation or the session is further delegated.

While indefinite redirection is relevant for fixed and reliable hosts, its design characteristics discussed above make it unsuitable for dynamic network environments such as P2P networks, Web services and ubiquitous computing, where the functionality of delegation would be particularly useful. Direct reconnection gives a robust and relatively efficient solution for such environments, and treats resources and failure in a manner that respects the logical conversation topology. For these reasons, we focus on designs based on direct reconnection in our implementation framework for delegation.

Reconnection strategy and asynchrony. The crucial element in the design of a reconnection-based delegation mechanism is its interplay with asynchronous communication (i.e. send is non-blocking). We illustrate this issue by revisiting the Travel Agency example in § 2 (see Figure 1): if Customer selects `ACCEPT`, Agency delegates to Service the active order session with Customer, and then Customer should send the `Address` to Service, its new peer. Now Customer, operating concurrently, may asynchronously dispatch the `Address` before or during the delegation, so that the message will be incorrectly delivered to Agency. We call such messages "*lost messages*": because Customer and Service may have inconsistent session states at the point of delegation, performing reconnection naively can break communication safety. Thus, a reconnection-based delegation requires careful coordination by a *delegation protocol* that resolves this problem.

Two reconnection-based protocols: key design ideas. Below we outline the key design ideas of the two reconnection-based protocols implemented in the SJ runtime. They differ in their approach to resolving the issue of lost messages.

Resending Protocol: (resend lost messages after reconnection) Here lost messages are cached by the passive-party and are resent to the s-receiver after the new connection is established, explicitly re-synchronising session state before resuming the delegated session. In Travel Agency, after the original connection is replaced by one between Customer and Service, Customer first resends the `Address` to Service before resuming the conversation.

Forwarding Protocol: (forward 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 Correctness Criteria for Delegation Protocols

For a delegation protocol to faithfully realise the intended semantics of session delegation, it needs to satisfy at least the following three properties. Below "control message" means a message created by the delegation protocol, as opposed to the actual "application messages" of the conversation.

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

Case 2: A has finished its side of s, and is waiting (in the separate close thread) for the acknowledgement.

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 case is called *simultaneous delegation*.

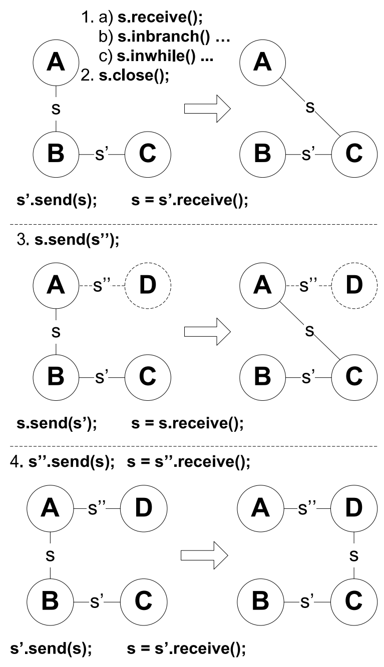

Fig. 4. The scenarios of session delegation.

P1: Linearity For each 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 lost messages can be correctly identified.

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

4.3 General Framework for Reconnection-Based Protocols

We first describe the structure shared by the two delegation protocols, introducing the common notation and terminology along the way. We write A for the passive-party, B for the s-sender; and C for the s-receiver. B will delegate the session s to C via s'. In each protocol, B will 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. Henceforth, we say "A" to mean the "runtime for A" if no confusion arises.

We assume each delegation protocol uses the same (TCP) connection for both application and control messages, as in our actual implementation. Since this means ordering is preserved between the two kinds of messages, the delegation

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

Fig. 5. Operation of the resending protocol for Case 1.

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 of s. The subsequent behaviour of the delegation protocols depends on what A was originally expecting this input to be. There are four cases, as illustrated in Figure 4 (the first picture corresponds to Cases 1 and 2, the second Case 3 and the third Case 4). Case 3 comes from the fact that delegating a session is a compound operation that contains a blocking input. 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 illustration, we return to Travel Agency: 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. The remaining three cases are outlined 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 5. The key feature of this protocol is the use of session types at runtime to track the progress of the two peers of the delegated session: this makes it possible to exactly identify the session view discrepancy ("the lost messages") and re-synchronise 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 $DS^B_A(C)$, to A. As stated, this signal contains the runtime session type of the session being delegated, from B's perspective, denoted ST^B_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

1.	$\mathsf{B}\rightarrow\mathsf{C}$:	"Start delegation"						
2.	C:	*open server socket on free port* p_{C}						
3.	$\mathsf{C}\rightarrow\mathsf{B}$:	p_{C}						
4.	$\mathsf{B}\rightarrow\mathsf{A}$:	$DS^{\mathsf{B}}_{\mathsf{A}}(\mathsf{C}) = \langle IP_{\mathsf{C}}, p_{\mathsf{C}}\rangle$						
5.	$\mathsf{A}\rightarrow\mathsf{B}$:	ACK_{AB}	5'.	B:	enter f/w mode			
6.	A:	close s	6'.	$\mathsf{B}\rightarrow\mathsf{C}$:	$\tilde{V}{::}ACK_{\mathsf{AB}}$			
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. 6. Operation of the forwarding protocol for Case 1.

(after receiving it) close their respective socket endpoints for the original session connection: any lost messages at B are simply 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 ...`!{ACCEPT:!<Address>}`, and the runtime type $ST^{\mathsf{A}}_{\mathsf{B}}$ (the view from A) is ...`?{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 6.

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 delegation 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 in 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 operate in parallel with two cross-dependencies, 6' on 5 and 8" on 7.

4.6 Outline of Delegation Cases 2, 3 and 4

We summarise how the two protocols behave for the remaining three cases of the description in § 4.3. The full protocol specifications are found at [26]. In both protocols, each of the remaining cases is identical to Case 1 in most parts: the key idea is that the role of the delegation acknowledgement ACK_{AB} in Case 1 is now played by some other control signal in each case.

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 to B (instead of ACK_{AB}) 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 $LM(ST_{\mathsf{B}}^{\mathsf{A}} - ST_{\mathsf{A}}^{\mathsf{B}})$ at Step 8 to C in order to start the subsequent run of the delegation protocol between 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_{AB} at Step 5, B receives $DS_{\mathsf{B}}^{\mathsf{A}}(\mathsf{D})$ from A. In the resending protocol, C then buffers the lost messages from A, closes this intermediate connection, and reconnects to the port at which D is waiting (C gets the address of D from A). In the the forwarding protocol, the behaviour is similar to that for *Case 3*.

4.7 Correctness of the Delegation Protocols.

Below we briefly outline the key arguments for the correctness of our delegation protocols with respect to the three properties **P1-3** in § 4.2, focusing on *Case 1* of the resending protocol. For details and the remaining cases, see [26].

Property **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 that deadlock arises only when a cycle (like $\mathsf{A}\rightarrow\mathsf{B}$ and $\mathsf{B}\rightarrow\mathsf{A}$) is executed concurrently, which is impossible from the protocol definition. Readiness holds since the connection to p_{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_{AB}. The key property for the correctness argument is **P3**. This holds since the sending action from C takes place after the lost messages from Step 8 are stored at C, which in turn holds since the sending action from C uses the same port p_{C}. Hence the order of session messages is preserved before and after the protocol.

5 SJ Runtime Performance

The current implementation of SJ [26] supports all of the features presented in this paper, including implementations of both the forwarding and resending protocols in § 4, called `SJFSocket` and `SJRSocket`. This section presents some performance measurements for the current SJ runtime implementation, focusing on micro benchmarking of session initiation and the session communication primitives. Although the current implementation is as yet unoptimised, these preliminary benchmark results demonstrate the feasibility of session-based communication and the SJ runtime architecture: `SJFSocket` communication incurs little overhead over the underlying transport, and `SJRSocket`, despite additional costs, is competitive with RMI. The full source code for the benchmark applications and the raw benchmark data are available from the SJ homepage [26].

The micro benchmark plan. The benchmark applications measure the time to complete a simple two-party interaction: the protocols respectively implemented by the 'Server' and 'Client' sides of the interaction are

```
begin.?[!<MyObject>]*              begin.![?(MyObject)]*
```

which basically specify that the Server will repeatedly send objects of type `MyObject` for as long as the Client wishes. Recall that session-iteration involves the implicit communication of a boolean primitive from the outwhile party (here, the Client) to its peer. Although this protocol does not feature branching, the SJ branch operations (communication of the selected label) are realised as the send/receive of an ordinary object (`String`), hence perform accordingly.

The implementation of these protocols in SJ is straightforward, and similarly for the "untyped" TCP socket implementation in standard Java (referred to as `Socket`), which mimics the semantics of the session-iteration operations using while-loops (with explicit communication of the boolean control value). `Socket` serves as the base case (i.e. direct usage of the underlying transport) for comparison with the SJ sockets. For the RMI implementation (`RMI`), the session-iteration is simulated by making consecutive calls to a remote server method with signature `MyObject getMyObject(boolean b)` (the boolean is passed to attain the same communication pattern). Henceforth, a session of *length* n means that the session-iteration is repeated n times; for `RMI`, n remote calls.

The benchmark applications specifically measure the time taken for the Client to initiate a session with the Server and finish the session-iteration. For `Socket`, session initiation simply means establishing a connection to the server. For `RMI`, the connection is established implicitly by the first remote call (RMI "reuses" a server connection for subsequent calls), but we do not include the cost of looking up the remote object in the RMI registry. Each run of a session is preceded by a dummy run of length one: this helps to stabilise the Server and Client processes before the actual benchmark run, and removes certain factors from the results such as class loading and verification. The `RMI` dummy run calls an instance of the remote object hosted on the local machine, to avoid creating a connection to the Server before the actual benchmark run.

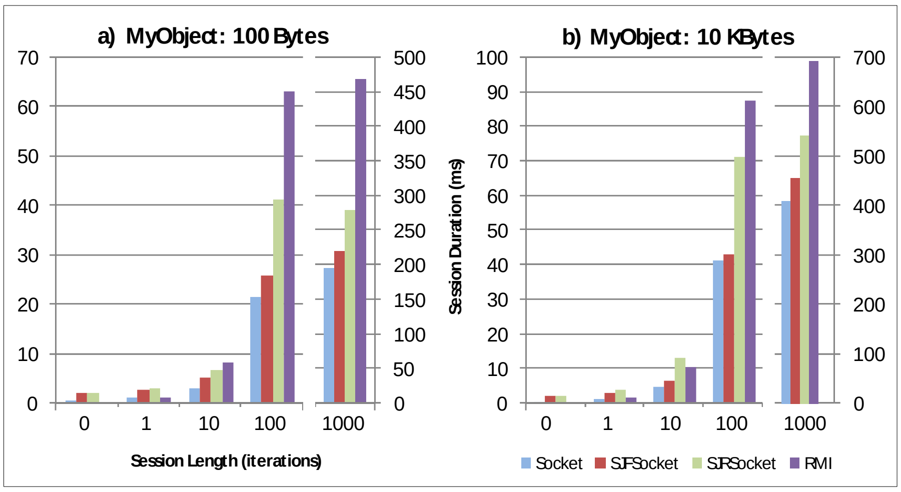

Fig. 7. Benchmark results for `MyObject` sizes 100 Bytes and 10 KBytes.

The benchmarks were conducted using `MyObject` messages of serialized size 100 Bytes (for reference, an `Integer` serializes to 81 Bytes) and 10 KBytes for sessions of length 0, 1, 10, 100 and 1000. We recorded the results from repeating each benchmark configuration 1000 times in low (∼0.1ms) and higher latency (∼10ms) environments. Nagle's algorithm was disabled (`TCP_NODELAY` is set to true) for these benchmarks, for both the standard and SJ sockets. `RMI` was run using the default settings for each platform. Runtime state tracking is disabled (as a default) for `SJFSocket`.

The low latency environment consisted of two physically neighbouring PCs (Intel Pentium 4 HT 3 GHz, 1 GB RAM) connected via gigabit Ethernet, running Ubuntu 7.04 (Linux 2.6.20.3) 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 0.10 ms. The higher latency benchmarks were recorded using one of the above machines as the Client (the timer) and a Windows XP PC (SP2, Intel Core Duo 3 GHz, 2 GB RAM) with an average communication latency (56 Byte ping) of 8.70 ms.

Results. We first look at the low latency results: minimising the latency factor emphasises the internal overheads of the SJ runtime. Graphs a) and b) in Figure 7 compare the results from the four benchmark applications over each session length for each of the two `MyObject` sizes.

The results show that `SJFSocket` exhibits low runtime overhead in relation to `Socket`, decreasing for longer sessions. Shorter sessions increase the relative cost of SJ session initiation, roughly 2ms in the measured environment. Note session initiation for both `Socket` and the session sockets involve establishing a TCP connection, but the SJ sockets do extra work to check session compatibility. `SJRSocket` is slower than `SJFSocket`: the cost for session initiation is the same for both, but `SJRSocket` employs (1) runtime state tracking and (2) a different routine for serialization and communication (in order to retain serialized messages for the sent message cache, § 4). In fact, comparison of `SJFSocket` and `SJRSocket` using sessions that communicate only primitive data types [26] show that most of the overhead comes from (2) with runtime state tracking incurring little overhead. Despite these additional overheads, `SJRSocket` performs better than `RMI` for longer sessions.

The results from the higher latency benchmarks [26] also support these observations. We note a few additional points. The cost of session initiation, which involves sending an extra message, increases accordingly. As before, the relative overheads of the SJ sockets become smaller for longer sessions, although higher communication latency appears to ameliorate these costs at a higher rate. This means that the latency factor dominates the overheads of SJ, from both internal computation and additional communication costs (such as SJ message headers and extra control messages), under the present benchmark parameters. Indeed, the differences in performance over the longer sessions are minimal.

Travel Agency benchmarks. Preliminary performance evaluation of session delegation has also been conducted based on the Travel Agency example (§ 2). We measured the time required to complete the transaction (10 iterations of the order-quote exchange and Customer always accepts the quote) from the point of view of Customer. For comparison, a purely forwarding-based (i.e. no reconnection) equivalent was implemented using standard sockets. Unlike the above benchmarks, we now include the time needed to create the SJ socket objects and close the session (in the SJ runtime, close involves spawning a new thread, § 4). The results for `Socket`, `SJFSocket` and `SJRSocket` using three machines in the low latency environment (same specifications) were 3.8, 7.3 and 9.1 ms respectively. We believe these figures are reasonable, given the overheads of delegation (extra control messages and coordination, the cost of reconnection, resending by `SJRSocket`, and others), and that the delegation mechanisms, like much of the current runtime implementation, are as yet unoptimised. Note that Travel Agency is in fact close to a worst case scenario for this kind of delegation benchmark: as discussed in § 4, reconnection-based protocols are advantageous when there are a substantial number of communications after the delegation and/or when the session is further delegated, especially if latency is high.

Potential optimisations. Firstly, many optimisations are possible exploiting the information on conversation structure and message types given by session types, including session-typed message batching (possibly based on the size of the underlying transport unit), the promotion of independent send types (pushing asynchronous sends earlier), tuning of I/O buffer sizes, and others. Knowing the expected message types in advance can also be used to reduce the amount of meta data (e.g. SJ message headers, class tags embedded by serialization) for basic message passing. Secondly, sessions peers can exchange extra runtime information at session initiation; for example, no delegation means `SJRSocket` would not need to cache sent messages. We also envisage situations where the duality check may only be needed whilst an application is being developed and tested: once the application has been deployed in some trusted environment (e.g. a company using an internal messaging system), this check can be disabled.

6 Related Work

Language design for session types. An application of session types in practice is found in Web services. Due to the need for static validation of the safety of business protocols, the Web Services Choreography Description Language (WS-CDL) is developed by a W3C standardisation working group [24] based on a variant of session types. WS-CDL descriptions are implemented through communications among distributed end-points written in languages such as Java or WS4-BPEL. [3, 6, 14] studied the principles behind deriving sound and complete implementations from CDL descriptions. Session types are also employed as a basis for the the standardisation of financial protocols in UNIFI (ISO20022) [28]. We plan to use our language and compiler-runtime framework as part of the implementation technologies for these standards.

An embedding of session programming in Haskell is studied in [18]. More recently, [22] proposed a Haskell monadic API with session-based communication primitives, encoding session types in the native type system. A merit of these approaches is that type checking for session types can be done by that for Haskell. Type inference for session types without subtyping has been implemented for C++ [7]. In these works, a session initiation (compatibility check) for open and distributed environments are not considered, and session delegation is not supported; type-safe delegation may be difficult to realise since their encoding does not directly type I/O channels.

Fähndrich et. al [11] integrate a variant of session types into a derivative of C♯ for systems programming in shared memory uni/multiprocessor (non-distributed) environments, allowing interfaces between OS-modules to be described as message passing conversations. Their approach is based on a combination of session types with ownership types to support message exchange via a designated heap area (shared memory): session communication becomes basic pointer rewriting, obtaining efficiency suitable for low-level system programming. From the viewpoint of abstraction for distributed programming, their design lacks essential dynamic type checking elements and support for delegation, and other features such as session subtyping. In spite of differences between the target environments and design directions, our works both demonstrate the significant impact the introduction of session types can have on abstraction and implementation in objected-oriented languages.

A framework based on F# for cryptographically protecting session execution from both external attackers and malicious principals is studied in [9]. Their session specifications model communication sequences between two or more network peers (*roles*). The description is given as a graph whose nodes represent the session state of a role, and edges denote a dyadic communication and control flow. Their aim is to use such specifications for modelling and validation rather than direct programming; their work does not consider features such as delegation or the design of a session runtime architecture.

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 [20] is a programming language based on the π-calculus with linear and polymorphic types. Polyphonic C♯ [2] is based on the Join-calculus and employs 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. The Concurrency and Coordination Runtime (CCR) [5] is a port-based concurrency library for C♯ for component-based programming, whose design is based on Poly♯.

Occam-pi [19, 29] is a highly efficient concurrent language based on channel-based communication, with syntax based on both Hoare's CSP (and its practical embodiment, Occam) and the π-calculus. Designed for systems-level programming, Occam-pi supports the generation of more than million threads for a single processor machine without efficiency degradation, and can realise various locking and barrier abstractions built from its highly efficient communication primitives. DirectFlow [17] 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 an embedding into a host language for defining data-flow between components. In both languages, typing of a communication unit larger than a series of individual communications or compositions is not guaranteed.

X10 [30] is a typed concurrent language based on Java, and designed for high-performance computing. Its current focus is on global, distributed memory, with sharing carefully controlled by X10's type system. A notable aspect is the introduction of distributed locations into the language, cleanly integrated with a disciplined thread model. The current version of the language does not include first-class communication primitives.

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

7 Conclusion and Future Work

There is a strong need to develop structured, higher-level and type-safe abstractions and techniques for programming communications and interaction. This paper has presented the design and implementation of an extension of Java to support session-based, distributed programming. Building on the preceding theoretical studies of session types in object-oriented calculi [8, 10], our contribution furthers these and other works on session types with a concrete, practical language that supports the wide range of session-programming features presented.

Communication safety for distributed applications is guaranteed through a combination of static and dynamic type checking. We implemented two alternative mechanisms for performing type-safe session delegation, with correctness arguments, and showed that session types can be used to augment existing features such as remote class loading.

We believe our language exhibits a natural and practical integration of session type principles and object-oriented programming. The session-programming constructs and accompanying methodology, based on explicit specification of protocols followed by implementation through these constructs (with static type checking), aid the writing of communications code and improve readability. Our experience so far indicates that session types can describe a diverse range of structured interaction patterns [6, 28]; further examples, such as file-transfer and chat applications, are available from [26]. Future work includes detailed analysis of how session-based programming can impact on the development of more complicated and large-scale applications. Integration of session-based programming with such languages as [17, 19, 25, 30] is also an open subject.

Preliminary benchmark results demonstrate the feasibility of session-based communication and our session runtime architecture. Static and dynamic performance optimisations that utilise session type information (see §5) are a topic for future investigation. Other interesting directions include the incorporation of transports other than TCP into the session runtime, possibly coupled with the design of alternative abstractions to (session) socket. Session parties can dynamically monitor messages received against the expected type according to agreed protocol: however, our current work does not yet tackle deeper security issues involving malicious peers [9].

This work and future directions, combined with theories from [3, 15], are being developed as a possible foundation for public standardisations of programming and execution for Web services [6] and financial protocols [23, 28].

Acknowledgements. We thank the reviewers and Susan Eisenbach for their useful comments, Fred Van Den Driessche for his contributions to the implementation, and our academic and industry colleagues for their stimulating conversations. The work is partially supported by EPSRC GR/T04724, GR/T03208, GR/T03215, EP/F002114 and IST2005-015905 MOBIUS.

References

1. Acute homepage. `http://www.cl.cam.ac.uk/users/pes20/acute`.
2. N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.
3. 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.
4. 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.

5. CCR: An Asynchronous Messaging Library for C#2.0. `http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime`.
6. W3C Web Services Choreography. `http://www.w3.org/2002/ws/chor/`.
7. P. Collingbourne and P. Kelly. Inference of session types from control flow. In *FESCA*, ENTCS. Elsevier, 2008. To appear.
8. 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.
9. 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.
10. 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.
11. 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.
12. S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.
13. 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.
14. 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.
15. K. Honda, N. Yoshida, and M. Carbone. Multiparty Asynchronous Session Types. In *POPL'08*, pages 273–284. ACM, 2008.
16. IETF. Mobility for IPv4. `http://dret.net/rfc-index/reference/RFC3344`.
17. 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.
18. M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.
19. Occam-pi homepage. `http://www.occam-pi.org/`.
20. 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.
21. Polyglot homepage. `http://www.cs.cornell.edu/Projects/polyglot/`.
22. M. Sackman and S. Eisenbach. Session types in haskell, 2008. draft.
23. Scribble Project homepage. `www.scribble.org`.
24. S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), March 2006.
25. J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. StreamFlex: high-throughput stream programming in Java. In *OOPSLA*, pages 211–228. ACM, 2007.
26. SJ homepage. `http://www.doc.ic.ac.uk/~rh105/sessionj.html`.
27. 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.
28. UNIFI. International Organization for Standardization ISO 20022 UNIversal Financial Industry message scheme. `http://www.iso20022.org`, 2002.
29. P. Welch and F. Barnes. Communicating Mobile Processes: introducing occam-pi. In *25 Years of CSP*, volume 3525 of *LNCS*, pages 175–210. Springer, 2005.
30. X10 homepage. `http://x10.sf.net`.