Type-Safe Communication in Java with Session Types

Raymond Hu   Nobuko Yoshida
Imperial College London
rh105@doc.ic.ac.uk   yoshida@doc.ic.ac.uk

Kohei Honda
Queen Mary, University of London
kohei@dcs.qmul.ac.uk

Abstract

This paper demonstrates the impact of integrating session types for communication behaviour into object-oriented languages, through their implementation in a distributed Java. Session types abstract a potentially unbounded structured sequence of communications in an easy-to-read type syntax, and guarantee the absence of communication errors through static type-checking. The introduction of session types not only contributes to clear and statically correct communication programs, but also enables the efficient, transparent delegation of an ongoing conversation and leads to the potential for significant performance improvement. To our knowledge, this is the first practical implementation of higher-order session types (delegation) with subtyping for an object-oriented language. The use of session types enables the decoupling of communication abstraction from its implementation mediated by type information: the runtime can encapsulate low-level communication mechanisms for their effective exploitation, with minimal overhead. The benchmark results show our implementation outperforms RMI, the standard API for typed inter-process communication.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.1.5 [*Programming Techniques*]: Object-oriented Programming; D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Theory, Design

Keywords communication, Java, session types, typing system, socket, NIO, protocols, delegation

1. Introduction

Communication is becoming one of the central elements of application development in object-oriented programming languages, ranging from web applications to high-performance computing in multiprocessor systems to corporate application integration to business protocols among organisations and individuals to core-to-core communication in multicore processors. A frequent programming pattern in such communication-centred applications arises when process interact via some structured sequence of communications, which as a whole forms the natural unit of a *conversation*. One-way message passing and RPC-style communication are simplest cases of such conversation patterns, but a conversation may also branch into multiple sub-conversations or even into loops.

One of the widely known programming abstractions through which such conversation patterns can be realised is the socket, available through APIs in many object-oriented languages such as Java and C♯. Socket-based programming (in particular for the widely used TCP-based sockets) offers a framework for representing communications via its two-way byte stream abstraction, typically abstracted as a *connection*. Although a suitable medium for the representation of conversational structures, socket-based programming suffers from several disadvantages.

- The byte stream abstraction is too low-level: no direct language-level abstraction is provided for what each chunk of raw data means, let alone the structure of a conversation as a whole.
- Control flows in a pair of communicating programs should together realise a consistent conversation structure: the lack of this abstraction means a programmer can easily fail to, for example, handle a specific incoming message or send a message at the correct time, with no way to detect such mistakes before runtime.
- The socket abstraction is directly coupled to a specific transport mechanism. Thus streams are tied to a physical connection, making delegation of an ongoing conversation difficult: optimisation using a raw interface affects source code, and running the same programs using different transport mechanisms is practically impossible.

These observations motivate the search for an abstraction mechanism for object-oriented programming which can naturally represent diverse conversation structures and be efficiently mapped to representative transport mechanisms, whilst preserving the familiar programming style of socket. Can we find such an abstraction? How can we specify a conversation structure concisely and clearly? Can we quickly validate whether a program conforms to a given specification? What may be the impact of this abstraction on efficiency?

Session types offer a method for abstracting and validating structured communication sequences. In session types, the specification of a conversation structure is given as a type, against which we can check whether a given program conforms to the specification. Session types have been studied in many contexts in the last decade, including π-calculus-based formalisms [8, 9, 22, 25, 26, 38]; multithreaded functional languages [24, 41]; Haskell [31]; Ambients [20]; CORBA [39]; operating systems [18]; and Web Services [11, 12, 37, 43]. In addition, recent studies on type-theoretic foundations of session types in object-oriented languages [14–17] have demonstrated that a clean integration of object-oriented idioms and session-based communication primitives is possible, backed up by rigorous theories. These studies use formalisms distilling the key elements of object-oriented languages for accurate analysis.

On the basis of these studies, the present work explores the potential of session-based idioms and their type disciplines as a practical basis for communication-based object-oriented programming, through their integration into Java. More concretely, we propose the design-implementation framework which consists of the following elements and examine its impact.

1. A type syntax for sessions based on [14–17], but with enhanced readability and conformance to Java syntax.
2. APIs and extended program syntax for session-based communication [14–17], with code mobility from [6].
3. Static type checking of programs with respect to session types at compiletime, implemented over the Polyglot compiler framework [33].
4. Dynamic compatibility validation at the time of session initiation through a handshake protocol (the compatibility check uses the session subtyping [23]).
5. A runtime layer which encapsulates a variety of communication mechanisms with minimal overhead, and which incorporates protocols for opening, delegating and closing sessions, abstracting from physical connections. This layer makes extensive use of session type information from point 1.

We have implemented all the above elements over Java (JDK 5.0) with two kinds of session-typed sockets. One of the central design decisions is to decouple programmers' description of communication actions (abstraction concerns) from their execution mechanism (implementation concerns), analogous to high-level control flows and the underlying machine instructions in the sequential structured programming. This principle leads to several significant consequences in design methodologies and performance:

- A new programming discipline/style for communication-based programs with guaranteed type and communication safety, which begins with a specification of intended communication structures using session types.
- Efficient, transparent delegation of a session from one party to another, abstracting conversations from physical connections.
- Communication performance that incurs very low overheads on the underlying transport, and which significantly improves over RMI, a representative method for typed inter-process communications.

For the first point, the programming style helps minimise the complexity of communication programming caused by synchronisation errors, providing both clarity and the safety guarantee. For the second point (delegation), the use of predetermined communication structures through session types is crucial for efficient, tractable implementation. Types also help simplify correctness arguments for the implementation.

The third point (performance) is made possible by combining the direct use of the untyped interface of each transport (tested for standard and NIO-based sockets) with predetermined type information, while exposing only a high-level, typed abstraction to programmers. Since all structural information is exchanged at the initial handshake, session communicationcan make good use of the underlying transport even before many potential optimisations. This is particularly visible when a session constitutes more than a few interactions, but even for a session consisting of a single request-reply, our implementation runs as fast as, and often much faster than, the equivalent RMI code.

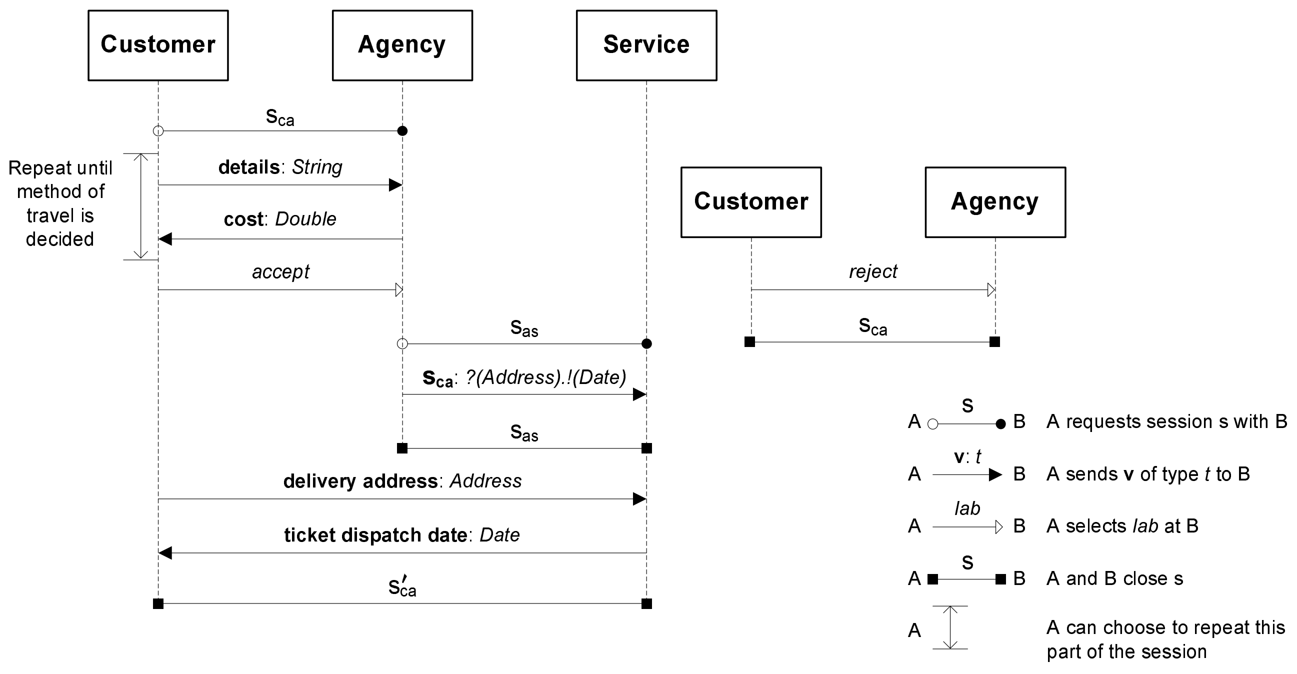
In summary, the present work clarifies, with a concrete implementation, the significant impact the introduction of sessions and session types into object-programming languages can give, on both programming discipline and performance, and suggests possible directions for further exploring their potential.

Structure of the paper Section 2 illustrates communications programming based on session types through examples. Section 3 details the type-checking mechanism. Section 4 summarises the main API and the translation to Java in our implementation. Section 5 illustrates the design and implementation of the accompanying runtime, with a focus on the delegation protocol. Section 6 reports the benchmark results. Section 7 discusses extensions and related work. Section 8 concludes the paper with further topics. The compiler, API, runtime and benchmark programs can be found at [4].

2. Programming with Session Types

This section illustrates the main features of the language extensions and programming methodology we propose through a simple example, an implementation of a ticket ordering system for a travel agency. The example comes from a use case in web services [10–12, 43], expressing typical collaboration patterns for business protocols.

Figure 2 depicts the scenario of interaction in this application. In words, it proceeds as follows:



Customer
Agency
Service
s_{ca}
Repeat until method of travel is decided
details: String
cost: Double
accept
s_{as}
s_{ca}: ?(Address).!(Date)
s_{as}
delivery address: Address
ticket dispatch date: Date
s'_{ca}
Customer
Agency
reject
s_{ca}
A s B A requests session s with B
A v: t B A sends v of 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

Figure 2.1. A ticket ordering system for a travel agency.

1. A customer (Customer) begins an *order session* s_{ca} with the travel agency (Agency) and requests and receives the price for journeys. The latter part may be repeated arbitrary many times, initiated by Customer.
2. After making a decision, Customer either accepts or rejects an offer from the agency; the two possible paths are illustrated separately in adjacent flows in the diagram.
3. If accepted (omitting such details as the handling of payment methods), Agency opens the session s_{as} with a travel service (Service) and it *delegates* the remaining interaction (at s_{ca}) with Customer to Service. The particular travel service to which Agency connects is likely to depends on the travel method chosen by Customer, but this logic is external to the present protocol.
4. Customer then sends a delivery address for the tickets, and Service replies with the date when the tickets shall be dispatched. The transaction is now complete.
5. If rejected in 2, the session promptly terminates.

One of the significant points in this example is support for *session mobility* through delegation. After the delegation, Customer continues to communicate in the same session, changing its partner from Agency to Service. The delegation is transparent to Customer. Many real-life applications involve similar patterns, ranging from server-client interactions to business protocols to message passing in parallel programs. As we shall see later, safety of such communication patterns can be ensured using session types.

Specifying Session Types We now turn this scenario into a program using session-based idioms and types, incorporated into Java syntax. The programming style we advocate for communication-centred programs starts from specifying the intended communication patterns as session types.

A session type encapsulates the structure of a conversation between two parties, abstracting away from such details as concrete message values. For example, the session type for Customer may be declared as:

```
protocol customerProtocol {
  begin.          // Commence session.
  ![              // Iteration starts:
     !(String).   // send a String, then
     ?(Double)    // receive a Double.
  ]*.             // Iteration ends.
  !{              // Select one of:
    ACCEPT: !(Address).?(Date).end,
    REJECT: end
  }
}
```

The keyword `protocol` signifies the start of a session type declaration. We can read this declaration informally as follows: the `customerProtocol` starts with a repeated interaction sequence (denoted by `![..]*`, illustrated later), consisting of sending a String (denoted by `!(String)`) and receiving a double precision integer (denoted by `?(Double)`). After the iteration, one of the two options, `ACCEPT` and `REJECT`, will be selected. If `ACCEPT`, communication proceeds by sending an address, receiving a date, and session terminates; if `REJECT`, the session terminates immediately.

Creating a Session-Typed Socket After declaring the required session types, we next create channels associated with these types. For familiarity, our API offers two abstractions close to standard socket programming in Java:

- *session-typed socket* (often *session socket* for short), of class `STSocket`, representing an endpoint of a conversation (corresponding to `Socket` in Java);
- *session-typed server socket* (often *server session socket* for short), of class `STServerSocket`, representing a port on which a server listens for connection requests (corresponding to `ServerSocket`).

These sockets differ from the standard sockets in that the session type is bound to the socket at creation. This is performed by the static `create` method of class `STSocket`. For example, Customer creates a session socket by:

```
STSocket agency =
  STSocketImpl.create(customerProtocol);
```

where `customerProtocol` is the session type declared on the previous page. Creating a server session socket is similar, but also binds the server socket to a particular port. For example, Agency's program will contain:

```
STServerSocket server =
  STServerSocketImpl.
    create(agencyProtocol, port);
```

where `agencyProtocol` is the session type which is *dual* to `customerProtocol`:

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

In brief, the dual session type is given by inverting the input `!` and the output `?`, see §3.

Opening a Session A session is started by connecting to a server using the `request` operation. As in ordinary sockets, the server is identified by an IP address and a port number, and so Customer commences a session with Agency by:

```
agency.request(Agent, port);
```

Concurrently, Agency may have:

```
STSocket customer = server.accept();
```

As in standard server sockets, the `accept` method listens for incoming connection requests and, when one arrives, creates a fresh session socket (`customer` above). Unlike standard server sockets, this handshake includes a validation that the type of the session requested by the client is *compatible* with that the server offers. This compatibility guarantees, based on inductive matching of types, that the subsequent interaction will not incur communication errors (details of the validation are discussed later in §3).

If this validation succeeds, the session is established: the client socket is now *active* (it was previously *inactive*), embodying the specified protocol. On the server side, a fresh session socket is returned, embodying the complementary protocol. If validation fails (e.g. if their communication structures do not match), the `accept` and `request` operations will throw a session incompatibility exception. Thus a session opening is typically enclosed within a try-catch statement (see [4] for a code sample).

In-session Communication (1): Send and Receive After establishing a session, typed messages (objects) can now be sent and received through the active session socket. Communications in a session, or *in-session communications*, are to be carried out precisely following the session type associated to the session socket. For example, `customerProtocol` says that the first two actions in the iteration should be `!(String)` and `?(Double)`. These types correspond to the `send` and `receive` operations provided by our API, with the obvious meaning. Thus, the first in-session communication by Customer should be:

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

describing the opening exchange by Customer, sending a travel detail and receiving its cost.

In-session Communication (2): Iteration Iteration of interactions is a common pattern found in many applications, including business protocols, application-level network protocols (e.g. SMTP), web interactions and parallel scientific computing. Following [16, 17], this is represented by two mutually dual types with syntax `![...]*` and `?[...]*`. Similarly to regular expressions, `[...]*` expresses that the interactions in `[...]` may be iterated zero or more times; the prefix `!` indicates the party who decides whether to iterate or exit the loop, and `?` the party which follows this decision.

As before, these type expressions have direct programming counterparts, `outwhile` and `inwhile` [16, 17], variants of the standard `while` statement. We illustrate them using examples. Recall `customerProtocol` starts with

```
![ !(String). ?(Double) ]*
```

This session type abstraction is manifest in the program using `outwhile`:

```
boolean notDecided = true;
... // Set travDetails
agency.outwhile(notDecided) {
  agency.send(travDetails);
  Double cost = agency.receive();
  ... // Set notDecided, change details
}
```

The `outwhile` command starts from a boolean condition for iteration, `notDecided`, whose truth value determines whether the loop continues or terminates, as in the standard `while` statement. The key difference is this decision is communicated (here, Customer) to the session peer (Agency), synchronising control flow at each party. If we view the structure of interaction as finite state automata (note session types are essentially regular expressions), then this implicit communication couples the FSAs of the interacting parties at appropriate states. Agency is programmed to behave dually:

```
customer.inwhile() {
  String travDetails = customer.receive();
  customer.send(cost);
}
```

Note `inwhile` does not have a boolean condition: the decision to iterate or exit is made by Customer and communicated to Agency before each iteration. The combination of `inwhile` and `outwhile` can be considered a distributed version of the standard while-loop.

These session constructs for iterative communication substantially reduce the design complexity of conversation. As a comparison, consider the following (arguably reasonable) implementation of this iteration at Customer in Java:

```
do {
  boolean bool = evaluate decision;
  outputStream.writeBoolean(bool);
  if(bool) {
    ... // Send travDetails, receive cost
  }
} while(bool);
```

where `outputStream` is a wrapper around the underlying TCP stream. As there is no syntactic construct that can explicitly capture the underlying control structure of interaction, it is easy for a programmer to make mistakes and the resulting code is hard to read. This issue is further exacerbated as communication protocols become more complex.

In-Session Communications (3): Select and Branch A session conversation may branch into sub-conversations with different interaction patterns. Since we need to maintain compatibility, session peers should be explicitly notified when a decision is made to enter a distinct sub-conversation. We have already seen the session type abstraction for this interaction structure in `customerProtocol`:

```
!{
  ACCEPT: !(Address).?(Date).end,
  REJECT: end
}
```

Selecting `ACCEPT` leads to a sub-conversation consisting of two communications; `REJECT` terminates the session. We expect Agency to behave dually, with the following type:

```
?{
  ACCEPT: ?(Address).!(Date).end,
  REJECT: end
}
```

The initial `?` indicates this is the party which (waits with options and) gets notified, as opposed to the party who selects.

As in the preceding cases, we have a pair of operations, `select` and `branch`, that correspond to this interaction structure. `select` may be considered similar to method invocation, while `branch` the object waiting with one or more methods. As an example:

```
if(want to place an order) {
  agency.select(ACCEPT) {
    agency.send(address);
    Date delivDate = agency.receive();
    agency.close();
  }
}
else {
  agency.select(REJECT) {
    agency.close();
  }
}
```

The value of the if-condition determines which option is selected, and hence should reflect whether Customer wishes to purchase tickets from Agency or not. `ACCEPT` and `REJECT` are called *branch labels*, or often simply *labels*. The corresponding code for Agency consists of:

```
customer.branch() {
  case ACCEPT: {
      ...
  }
  case REJECT: {
      customer.close();
  }
}
```

(The body of `ACCEPT` is given when we discuss delegation.) Note that the branch labels in the code are precisely those specified by the session type. We again contrast our session code against an implementation in standard Java:

```
switch(inputStream.readInt()) {
  case ACCEPT: { ...; break; }
  case REJECT: { break; }
  default: { ... }
}
```

Here, branch labels are assumed to be encoded as integers; otherwise, a more cumbersome if-else statement is needed. Without the guarantee that the received switch value will indeed be a valid label, a default case should be included, requiring more code (which session types render redundant) and impairing readability.

Session Delegation When Agency delegates the session to Service, Agency relinquishes that session (i.e. the session is closed), and it is up to Service to finish the remainder of the conversation with Customer according to the expectations of the latter. By imposing this requirement on Service, the conversation structure that Agency can offer to Customer remains the same whether there is delegation or not. In this way, we obtain both flexibility and safety.

Following our scenario, such a delegation is performed by Agency after ACCEPT is selected by Customer who proceeds by sending the delivery address and waits to receive the date for dispatch. At this point, the session has been delegated by Agency to Service with whom Customer actually interacts but without knowing. Since the delegation itself concerns only Agency and Service, it is only in the session type shared by these two parties (and their respective programs), as opposed to that of Customer, where the delegation is visible. The interaction between Agency and Service, from the perspective of the former, is specified as follows:

```
protocol delegationProtocol {
  begin.
  !(?(Address).!(Date).end).
  end
}
```

The delegationProtocol represents a single sending action which communicates (a contract for) the separate interaction structure ?(Address).!(Date).end to the receiver. Here, the delegation action is abstracted as a *higher-order session type* [16], where the specified message type is itself a session type, ?(Address).!(Date).end (this type denotes the stage of interaction reached by the delegated session at the point of delegation: the party that receives the session will resume the conversation from that stage).

In terms of actual programming, delegation manifests as the act of sending the session socket representing the session being delegated; communication of normal messages and higher-order values is uniform. For example, Agency delegates the session with Customer to Service by:

```
case ACCEPT: {
    STSocket delegate =
  STSocketImpl.create(delegationProtocol);
    delegate.request(Service, port);
    delegate.send(customer); // Delegation
    delegate.close();
}
```

When ACCEPT is selected, the agency requests a session of type delegationProtocol with Service, “sends” the customer session socket (transfers to Service the capability to converse in the current session with Customer), and then implicitly closes the socket (relinquishes its own right to converse). This action maintains session *linearity*: session membership is strictly for two mutually dual parties at all times. The dual session type to delegationProtocol is:

```
protocol delegatedProtocol {
  begin.
  ?(?(Address).!(Date).end).
  end
}
```

and the corresponding program for Service is:

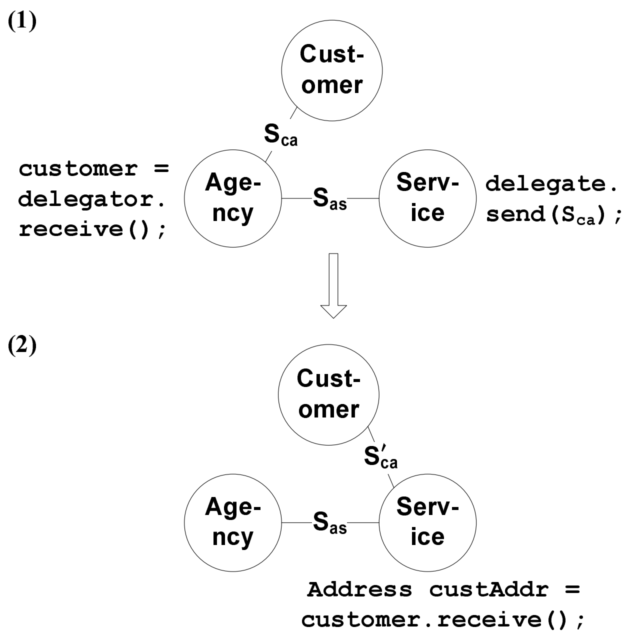

(1)
Cust-omer
S_{ca}
customer = delegator. receive();
Age-ncy
S_{as}
Serv-ice
delegate. send(S_{ca});
(2)
Cust-omer
S'_{ca}
Age-ncy
S_{as}
Serv-ice
Address custAddr = customer.receive();

Figure 2.2. Change in connection topology by delegation.

```
STServerSocket service =
  STServerSocket.
    create(delegatedProtocol,port);
STSocket delegator = service.accept();
// Receive delegated session
STSocket customer = delegator.receive();
service.close();
```

Service will receive and complete the session:

```
Address custAddr = customer.receive();
... // Calculate dispatch date
customer.send(dispatch date);
customer.close();
```

These operations simply dual to those performed by Customer after selecting ACCEPT and up to the end the session.

This example demonstrates how typed session enables safe, controlled evolution of connection topology, as illustrated in Figure 2. Prior to delegation (1), the sessions s_{ca} and s_{as} reflect the original connection configuration for each session. However, the delegation operation closes the connection underlying s_{ca}, and creates a new connection between Customer and Service; the relocated session is now termed s'_{ca} (2). Then the remaining interaction for session s'_{ca} is conducted over the new connection.

Agency can guarantee to Customer that the session will be safely completed (discounting e.g. I/O errors), despite its only fulfilling part of the session contract, because Service guarantees to Agency a completion of the delegated session. Meanwhile, the delegation is fully transparent to Customer.

The full code is listed in Appendix A.

3. Session Verification

The communication idioms for sessions presented in the previous section not only make the flow of communication behaviour more explicit and understandable, but also enable static verification of program conformance with respect to declared session types. When a program is actually executed, safety of the interaction depends not only on the local code but also the implementation of its peer: it may be the case that two communication peers do not own compatible interaction structures. Thus safety of interaction demands validation at runtime, which uses type information obtained from the static validation. These verifications lie at the heart of safe communication-centred programming with sessions.

3.1 Static Verification

General Idea The static session verifier checks that, for each session socket, whether the communications through it conform to the associated session type. Based on the preceding studies on session types [16, 26], our implementation relies on the following two elements.

- *Linearity* in the use of a session socket: there can be at most one owner of a session socket at one time.
- *Structural Correspondence* between communication actions and session types: session communication and the associated type structure should strictly match, mediated by this correspondence.

Linearity For ensuring linearity, we first demand that session sockets cannot be aliased: the initialisation of a `STSocket` reference should be through one of the three following methods: invoking `create` on a concrete implementation of `STSocket`, performing `accept` on a session server socket, or performing `receive` on a session socket "catch" a delegated session. A reference of `STSocket` type is treated as `final` except that a field of this type may be declared null and later assigned within a method body.

As an instance of the treatment of linearity, consider the following if-statement without an else-clause:

```
if(...) { s.send(...); ...}
```

The statement intuitively violates linearity because the session peer has no way to determine whether or not the send has been/will be executed [26]. However, if the whole session is completed within the if-statement, then linearity is no longer a problem:

```
if(...) {
  STSocket s = STSocketImpl.create(...);
  s.request(...);
  ... // Body of the session
  s.close();
};
```

Now the opposing party will have either one whole session communication or none; hence, unlike the previous case, we have no uncertainty as to the structure of a session. Also, if both branches of a conditional have a common supertype by subsumption, they can include session expressions with free sessions [11, 23]. Session subtyping shall be discussed in §3.2.

`begin`	`s.request/accept(...);`
`end`	`s.close();`
`?(`*`type`*`)`	`s.receive();`
`!(`*`type`*`)`	`s.send(obj); //obj is of type` *`type`*
`?{...}`	`s.branch() { ... };`
`!{...}`	`s.select(...) { ... };`
`?[...]*`	`s.inwhile() { ... };`
`![...]*`	`s.outwhile(...) { ... };`

Table 1. Session operations and corresponding types.

Structural Correspondence The static verification uses the direct correspondence between communication primitives and session type constructors, listed in Table 3.1.

For example, suppose a client session socket, say `s`, is declared with the following type:

```
begin.!(String).?(Int).end
```

Then the verifier searches for `s.request(...)` tracing the subsequent control flow: when it is found, the verifier clears the first construct of the session type, `begin`; it then searches for what corresponds to `!(String)`, i.e. the sending of a String via `s`. In this way, the verifier traces communication actions at `s` until the session is completed according to the session type, i.e. until the specified contract be fulfilled. After the session completes (`end` in the type, `s.close()` in the program), the socket `s` should no longer be used.

Based on this strict correspondence, the principles for the validation algorithm are cleanly articulated, though there are several subtle cases. One such instance is method invocation, for which we allow session sockets to be passed as parameters. This facility demands programmers to specify the expected session type of the STSocket, (rather than STSocket itself) as the type of the parameter. For example, we may declare a method `foobar` as follows. Let `p` below be `?(String).!(Int).end`.

```
... foobar(protocol p s, ...) {
    // completes 's' following p.
}
```

Note `p` above does *not* start with `begin` but ends with `end`: this method should receive an active session socket (i.e. one whose session has been established) as a parameter. The body of `foobar` should complete the specified session, while the caller should relinquish its ownership. Session sockets should not be passed to a method for parameter types other than an explicit session type.

Session delegation which induces dynamic evolution of connection topology (cf. §2), is treated similarly. Consider:

```
s1.send(s2); ...
```

The session type for s1 may have the shape:

```
begin...!(p)...end
```

where p is a session type fragment which ends with end (such as !(String).?(Int).end). The program should relinquish s2 after the above delegation (so it cannot be used after s1.send(s2);): the control is handed over to a receiving process, which is obliged to carry out the remaining session, represented by p. This determined the type discipline for the receiving side. Since a receiving process carries out this session, the above program in turn is regarded as fulfilling its contract for s2, including p, so that it becomes typable under the session type declared for s2.

There are two further interests for typing a delegation. First, the real type of a communicated session can be a *subtype* of the declared session type p (session subtyping will be explained in §3.2), which also holds when passing a session by method call. Second, delegation should not occur in the body of iterative constructs, including the standard while-statement, inwhile and outwhile. An exception is, as we saw in the if-statement, when the session is created and completed within the body of a loop. Intuitively, delegation is the act of giving away a session, so it does not make sense to do it more than once. These disciplines follow [16, 26].

Finally, since the session type declaration is separate from the actual communication actions, it sometimes helps to explicitly cast the received value with the type declared in the session type:

```
ClassA x = (ClassB)s.receive();
```

where ClassB is the class declared in the session type for s and which, therefore, is inferable (and in fact is automatically inserted by the compiler if omitted by the user). This style aids the clarity of the program text especially when, for example, ClassA is a superclass of ClassB, though it is formally unnecessary.

3.2 Dynamic Verification

General Idea Assume we have two programs in distinct distributed locations and each program has passed the static session verification. When these two programs, call them Alice and Bob, are going to interact, both may wish to know if they can carry out their conversation smoothly. Since Alice and Bob are distributed, they need to *communicate* to check compatibility between their interaction behaviours. Under the assumption that each program has been statically session-verified, there is no need to directly look at their code: Alice and Bob can use accurate abstractions of their communication behaviours, namely session types.

In our implementation, the party who does request sends her session type to the party who does accept. If compatibility is verified, the latter notifies the former, and a session commences. If not, again the requester is notified of the result, and the session aborts. The notion of compatibility is based on session subtyping from [11, 23], making verification more flexible and practical, without sacrificing session safety.

Compatibility We illustrate the key ideas behind compatibility validation with an example. Suppose Alice is verified to have the following type as a client at some port:

```
protocol aliceProtocol {
  begin.
  !{
    COFFEE: ?(Integer).end,
    TEA: ?(String).end
  }
}
```

And Bob has the type for a service port:

```
protocol bobProtocol {
  begin.
  ?{
    COFFEE: !(Integer).end,
    TEA: !(String).end,
    WATER: ?(Integer).end
  }
}
```

Then we observe the following:

- Alice's type says that she may ask for either COFFEE or TEA, and she is ready to receive an Integer (if COFFEE) or a String (if TEA).
- Bob's type says that he is ready to be asked for COFFEE, TEA *or* WATER, and will send an Integer or a String, or receive an Integer.

Alice's corresponding program may have the shape:

```
if(sleepy) {
  s.select(COFFEE) {..};
}
else }
  s.select(TEA) {..};
}
```

whereas Bob's could have the shape:

```
s.branch() {
  case COFFEE: { ... }
  case TEA: { ... }
  case WATER: { ... }
}
```

From these observations, we conclude that Alice and Bob's conversation will go through safely. Then Bob will answer "yes" to Alice, establishing the session, completing the initial handshake: Alice and Bob can now proceed to interact.

The compatibility used above intuitively says that if, at each stage of conversation, there are the same or more options at the branching side (?) than what may potentially be selected at the selection side (!), then an interaction is compatible. A key formal rule for compatibility follows:

```
void request(String host, int port)
  throws IOException,UnknownHostException,
  STIncomaptibleSessionException
Object receive() throws IOException,
  ClassNotFoundException
void send(Object obj) throws IOException
branch() throws IOException
select(String lab) throws IOException
inwhile() throws IOException
outwhile(boolean bool) throws IOException
void close() throws IOException
```

Figure 4.1. Session API

If each $\mathtt{p}_i$ is compatible with $\mathtt{q}_i$ ($1 \leq i \leq n$), then $!\{\mathtt{l}_1\mathtt{:p}_1\mathtt{,}\ \mathtt{..,}\ \mathtt{l}_n\mathtt{:p}_n\}$ is compatible with $?\{\mathtt{l}_1\mathtt{:q}_1\mathtt{,}\ \mathtt{..,}\ \mathtt{l}_n\mathtt{:q}_n\mathtt{,}\ \mathtt{...,l}_{n+m}\mathtt{:q}_{n+m}\}$ ($m \geq 0$).

For other rules, we use duality (defined by exchanging input `?` and output `!`, e.g. `!(Int)` and `?(Int)` are dual: `end` and `end` are also dual). The above rule comes from session subtyping [11, 12, 23], whose rule for selection is:

If each $\mathtt{p}_i$ is a subtype of $\mathtt{q}_i$ ($1 \leq i \leq n$), then $!\{\mathtt{l}_1\mathtt{:p}_1\mathtt{,}\ \mathtt{..,}\ \mathtt{l}_n\mathtt{:p}_n\}$ is a subtype of $!\{\mathtt{l}_1\mathtt{:q}_1\mathtt{,}\ \mathtt{..,}\ \mathtt{l}_n\mathtt{:q}_n\mathtt{,}\ \mathtt{...,l}_{n+m}\mathtt{:q}_{n+m}\}$ ($m \geq 0$).

4. API and Compilation into Java

This section briefly summarises the STSocket API for session socket programming, and illustrates the source-to-source translation of session constructs into Java by our compiler.

API for Session-Typed Socket The STSocket API, illustrated in §2, is summarised in Figure 4.

While session-based operations can be represented in many different ways, we have decided on the style of standard method invocation for a smooth integration into Java. `request` has similar semantics to the corresponding action of standard sockets. Message types carried in communication operations (`send` and `receive`) are primitive types, classes, interfaces, array types, and higher-order session types (for session delegation). Branch and select elements contain at least one case, although cases may be empty. `branch` and `select` use alphanumeric labels. `branch` is similar to a switch-statement but operates on a String value, has separate scope for each case, has implicit breaks, and has no default case. As we have observed, session types can be used as formal parameters to methods allowing passing of session end-points via method call. `select`, `branch`, `inwhile` and `outwhile` are method invocations with a trailing block statement. We find this extension necessary, especially for the last three, in order to make conversation flows explicit, enhancing readability.

For explaining this point, we briefly illustrate their translation below.

Compilation into Java Our compiler is built on the top of Polyglot 1.3.4 framework [33], which translates the extended syntax into Java. At compilation time, we also perform the static session verification discussed in §3. The compiler consists of approximately 6KLOC (excluding comment lines and blank) in extension to the base Polyglot framework. It first verifies typability using the extended syntax, then (partly) using the extracted type information, translates the extended syntax and session type annotations into Java.

We first consider a `branch` operation:

```
c.branch() {
   LAB1: { ... }
   ...
   LABn: { ... }
}
```

which becomes, after translation:

```
String _branch0 = (String)s.branch();
if (_branch0.equals("LAB1")) {
  ...
} else if (_branch0.equals("LABn")) {
  ...
} else {
  // throw exception.
}
```

where the `branch` at the level of Java simply returns a string. A `select` is simply translated into the sending of a string.

The translations of `outwhile` and `inwhile` insert the communications implicit in these commands. First,

```
sess.outwhile(boolean expr){ ... }
```

becomes, after translation,

```
while(sess.outsync(boolean expr)){ ... }
```

where `outsync` is a helper method which evaluates the boolean expression once and sends its result through a session socket. Dually:

```
sess.inwhile() { ... }
```

becomes, after translation,

```
while (s.insync()) { ... }
```

where `insync` receives a single boolean value through the session socket and returns it.

Beside these translations, the compilation stage also mediates the difference in the type information between the session-based extension and Java. First, we insert the casting for each receive operation:

```
ClassA x = sess.receive();
```

becomes

```
ClassA x = (ClassB)sess.receive();
```

where `ClassB` is the type specified in the session type declaration for `sess` (note this can be different from `ClassA` as far as Java allows it).

Next, when a session type is mentioned in e.g. a method call, this should be translated into an STSocket type to reflect the actual data type being received: for cross-class compilation purposes, however, the session type information needs to be retained. Thus a method declaration:

```
retType method(protocol p s, ...) { ... }
```

is translated to, using a helper class `STProtocol` and an auxiliary variable used for cross-module verification `_STP_meth_param_0` associated with this method:

```
STProtocol _STP_meth_param_0 = p;
retType meth(STSocket s, ...) { ... }
```

5. Runtime for Session Communication

This section presents the design and implementation of the runtime layer. The main purpose of this layer is to efficiently encapsulate the underlying transport mechanism whilst maintaining the level of abstraction presented by our API. Thus, the key functions of session-based communications such as transport of messages and the protocols for opening, delegating and closing sessions are all handled internally within this layer. Most importantly, the runtime depends on session type information obtained from the type verification stage to perform these functions. Our current, initial implementation of the runtime layer, which includes the API, session sockets and support classes, consists of around 1 KLOC of Java.

5.1 General Framework

There are three basic functions in the present runtime layer, divided by the life-cycle of a session.

(1) (session-opening protocol) The opening of a session, which obtains and validates type information.

(2) (message delivery) The delivery of messages between the endpoints, using the type information of (1).

(3) (session delegation/closing protocol) Special operations on sessions including session delegation and the closing of sessions, again using the type information of (1).

5.2 Session-opening Protocol

The key idea is as discussed in §3.2. Assuming the underlying transport can send an untyped frame (byte array) reliably, the protocol exchanges session types and verifies their compatibility. The current implementation uses the following protocol, with **C** requesting to open a session with **A**:

$$
\begin{array}{lll}
1: & \mathbf{C}\rightarrow\mathbf{A}: & s\_type_C \\
2: & \mathbf{A}\rightarrow\mathbf{C}: & match(s\_type_A,\ s\_type_C).
\end{array}
$$

where $match(s\_type_A, s\_type_C)$ returns true or false depending on whether or not s_type_A and s_type_C are compatible. If true, a session is established, and message passing may proceed according to the designated type. If not, an exception is raised at both sides by the runtime layer.

5.3 Message Delivery

We assume a reliable transport in which communication is asynchronous (i.e. send does not block) and message order is preserved. Most transports support transfer of untyped frames (i.e. messages are byte arrays of specific length). Once we have this mechanism, we can send and receive messages, and expose them as typed objects. In the present implementation, we have experimented with (1) object stream via socket API and (2) socket channel via NIO. In the former we utilise implicit (de)serialisation and framing implemented in the API. Because of the session type information, the receiving party already knows the type of the received object, and can safely cast it. For the latter, we manually perform (de)serialisation, with explicit framing (which includes the size information of the serialised object, to handle the reception of a partial/multiple frames).

5.4 Session Closing Protocol

Closing a session differs from closing a TCP connection. First, the lifetime of a session depends on its type: a session may only be closed when the whole conversation is complete. Secondly, an additional handshake is required to support session delegation. These elements are elaborated when we present the concrete closing protocol along with the delegation protocol.

5.5 Session Delegation Protocol

Delegation and Asynchrony We assume the following two conditions for our underlying transport, in order to realise the structured communication with efficiency.

- *Communication is asynchronous* (ASYNC), i.e. no wait at senders
- *Message order preserving* (MOP), i.e. the order of two messages between two parties is preserved.

These are offered by the representative transport mechanisms such as TCP and SCTP. They however have a subtle interplay with session delegation. To illustrate, we revisit the example in §2, focussing on its delegation part, depicted in Figure 2: if Customer selects `ACCEPT`, Agency delegates the session to Service, who then picks up the conversation with Customer where Agency left it off.

Our focal point is when Agency is ready to delegate his session with Customer, to Service. Meanwhile, Customer, unaware of the intentions of Agency (and the existence of Service), may naturally proceed by sending the delivery address to *Agency* and waits for the dispatch date: though the address should have been received by Service. Under (ASYNC), any delegation scheme cannot avoid such misdirected messages. An effective delegation protocol therefore should handle such lost messages in some way.

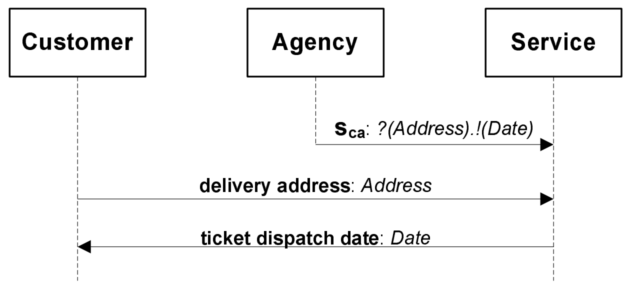

Customer
Agency
Service
s_{ca}: ?(Address).!(Date)
delivery address: Address
ticket dispatch date: Date

Figure 5.1. Delegation by Agency to Service.

1 :	**S**:	Creates a server socket at $port_s$
2 :	**S**→**A**:	$port_s$
3 :	**A**→**C**:	$ST^a(s_{ca})$, IP_s, $port_s$
4 :	**A**:	Spawns a thread to close s_{ca}
5 :	**C**:	Spawns a thread to close s_{ca}
6 :	**C**:	Connects to $IP_s : port_s$
7 :	**C**→**S**:	"Lost messages": ST$^c(s_{ca})$ - ST$^a(s_{ca})$

Figure 5.2. Delegation Protocol

Delegation Protocol At the heart of the delegation protocol is the use of session types to determine the discrepancy between the view of a conversation from party. This discrepancy reveals which messages may have been lost, due to communication asynchrony, during the relocation of the delegated session. The protocol is given in Figure 5.2. We use the example of §2 for explanation, but the design of the protocol is general for all (single) delegation situations. We refer to the *runtime* at each peer by **C** (Customer), **A** (Agency) and **S** (Service). "**S**→**A**: V" denotes that **S** sends V to **A**. $port_s$ is a fresh (unbound) port at **S**, and IP_s is the IP address of **S**. $ST^a(s_{ca})$ is the *current session type* of **A** in (i.e. A's view of) the session s_{ca}, representing the stage of the session that **A** has reached immediately prior to the delegation; this is known at compile-time. In Line 7, $ST^c(s_{ca})$ is the type representing the stage which **C** has reached in s_{ca} just before Line 4, which cannot be predicted statically due to asynchrony. $ST^c(s_{ca}) - ST^a(s_{ca})$ is illustrated below. We explain each step of the protocol line by line.

1. **S** binds a new `ServerSocket` to $port_s$, for receiving the delegated session.
2. **S** sends the value of $port_s$ to **A**.
3. **A** informs **C** of the delegation by sending the specified triple.
4. **A** spawns a thread to close s_{ca} (a *closing thread for* s_{ca}) asynchronously with Lines 5–7. The protocol that closing threads follow is given below.
5. By Line 3, **C** knows s_{ca} should be closed, and likewise spawns a closing thread for s_{ca}.
6. Using the values received in Line 3, **C** connects to **S**.
7. **C** calculates the "difference" between $ST^c(S_{ca})$ and $ST^a(s_{ca})$ (the latter obtained in Line 3) and sends the corresponding "lost messages" to **A**.

The protocol relies on session types in several places:

- Line 1: **S** creates a fresh server socket since he knows that delegation is to take place at this point *a priori*. Similarly, in Line 2, no need for **A** to notify it wishes to do delegation, since **S** already knows of it.
- Lines 4 and 5: the original session between **A** and **C** can be immediately closed, because only terminating session types (ending with `end`) are allowed as carried types, as discussed in §3. This condition is required for type soundness of session typing [16, 26].
- Line 7: session type information is used to determine which message(s) need to be resent.

We illustrate the last point, on Line 7, using our example: $ST^c(s_{ca})$ is given by

...!{`ACCEPT:!(Address)`}

while $ST^a(s_{ca})$ is

...?{`ACCEPT:` },

thus the difference is `!(Address)`. Intuitively, **C** compares the stage it has reached against the stage **A** has reached, to determine which messages need be resent to **S** for maintaining consistency of the session. Since **S** will continue the delegated session from the position specified by $ST^a(s_{ca})$, **C** can synchronise with **S** by resending those message(s), in this case the delivery address. Note this protocol requires that the runtime of a session peer store copies of sent messages (up to the previous receipt of a message). As shall be discussed in §6, this little affects communication efficiency. Furthermore, due to session types, the space overhead (the number of cached messages) at each stage is precisely predictable.

Finally there are two subtle cases which require scrutiny.

(a) (double delegation) In Line 3 of Figure 5.2, if **C** is simultaneously delegating s_{ca} to, say, **U**, then this session should be relocated to encompass move **S** and **U**.

(b) (closing and delegation) If **A** in Figure 5.2 delegates the session whose remainder is solely the sending actions from **C** to **S**, then **C** may have sent out all her messages and, while she is closing the session, may receive the message in Line 2 from **A** (Line 4 becomes redundant).

Below we discuss these cases briefly. See [4] for details.

Double Delegation Suppose **C**, when she receives the message in Line 3, is ready to perform a delegation. Then she will eventually receive a fresh port from **U** (corresponding to Line 2 of Figure 5.2). By the shape of a session type, only one of **C** and **A** has "lost messages", if any.

1. If **C** has one or more lost messages, then **C** sends them to **S**, and then informs **S** to reconnect to **U**. **U** and **S** can then resume the session. Similarly for the symmetric case.
2. If neither has such messages, then we choose one of **U** and **S** (by any consistent method e.g. the delegation target of the original session requestor), say **U**. Then **U** reconnects to **S**, and the session is resumed.

This protocol again depends on session type information. Figure 5.5 describes the case (1). In the intermediate stage, s'_{ca} is a link to send the lost messages. In the final stage, the new session s''_{ca} replaces the original s_{ca}.

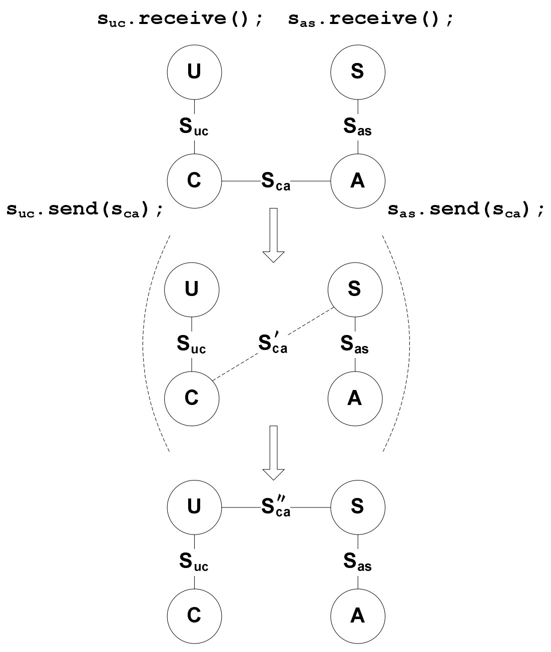


Figure 5.3. A double delegation scenario.

Closing Protocol Finally we present the closing protocol. Because a session party should be ready to receive a delegation message any time, possibly after reaching the end of the protocol, we conduct this protocol in a new thread (cf. Lines 4/5 of Figure 5.2); Once a logical conversation is complete, a session peer should be able to proceed without waiting for the closing protocol to finish.

We briefly explain the protocol in Figure 5.4. Lines 1 and 2 may occur asynchronously. Depending on the content of *rep*, we move to Lines 3a or 3b. Line 3a is when the other party also wishes to close the session. In Line 3b, DELEGATE denotes the message in Line 3 of Figure 5.2. Line 3b has two cases:

- If the closing thread is the result of session delegation (e.g. as for the closing thread created by **A** in Line 4 of

1 :	**X**$\rightarrow$**Y**:	SESSION_END
2 :	**Y**$\rightarrow$**X**:	*rep*
3a :	**X**:	*rep* = SESSION_END: close connection, exit.
3b :	**X**:	*rep* = DELEGATE: do a delegation, then goto 2.

Figure 5.4. Session Closing Protocol

  Figure 5.2), then both **X** and **Y** wish to perform delegation and commence the double delegation protocol.
- If not, **X** performs the delegation protocol of Figure 5.2.

6. Performance Evaluation

The runtime discussed in §5 has been implemented using two methods, the standard socket (referred to as STSocket) and the channel-based socket of `java.nio` [29] (STNIO). This section reports benchmark results for these two implementations. Our main aim is to understand the overhead our implementations add to the underlying communication mechanisms; and how they compare to RMI, as a representative API for typed communication in Java.

All benchmark programs can be found in [4].

Two Implementations: Socket and NIO STSocket uses `ObjectInputStream` and `ObjectOutputStream` to read and write to TCP streams. In contrast, STNIO uses block-based I/O through non-blocking `SocketChannels`, performing serialisation and in-channel message framing manually. Each runtime incorporates mechanisms for the features discussed in §5, including: the session initialisation, session type tracking and caching copies of sent objects to support delegation, and session closing threads. In both implementations, we set TCP_NODELAY to true, disabling Nagle's algorithm (otherwise TCP suffers severe latency).

Because of relatively high abstractions in the STSocket implementation, the caching of sent objects for delegation (cf. §5.5) in STSocket is done by duplicating serialisation. For STNIO, however, we simply retain the reference to a serialised object binary, created manually.

Benchmark Programs We measure the performance of our implementations through a suite of client-server programs, written using STSocket, STNIO, their untyped counterparts (which we call *plain socket* and *plain NIO* respectively) and RMI, that observe the same simple protocol: Its client-side session type is given as:

```
begin.![ ?(MyObject) ]*.end
```

where `MyObject` is a wrapper for a byte array, allowing us to evaluate performance for differing message sizes. The corresponding STSocket (and STNIO) code has the following structure, where `repeat` is an integer parameter:

```
s.outwhile(repeat-- > 0) {
    MyObject mo = s.receive();
}
```

Note the communication of a boolean value (the result of evaluating the loop condition) is implicit.

This choice of protocol reflects one of the basic patterns of sessions, a sequence of reciprocal interactions. It also permits direct comparison with RMI. When the number of iterations is set to *one*, we say that the session is of length *one*, equivalent to a single RMI call. Longer sessions, counted in the same way, are given by increasing the number of iterations. Note a session of length n in the present terminology involves n round trips between two endpoints; the corresponding RMI code makes n-consecutive remote calls.

For controlled comparison, plain socket and plain NIO are obtained by simply removing all code relating to session types (as noted above) from STSocket and STNIO, leaving only the underlying communication infrastructure.

Measurement Method Each of our experiments measures the time required to complete a series of sessions of increasing length. We conducted the experiment twice for `MyObjects` of serialised size 100 bytes (called *small message*; for reference, a serialised Integer is 81 bytes) and 10 kilobytes (*large message*). For each run of the experiment (i.e. each combination of session length and message size), the benchmark was repeated until session completion time stabilised (typically to two significant figures), and the mean of the subsequent 20 executions was recorded.

The experiments 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 1.5.0_3. Latency between the two machines was measured using ping (56 bytes) to be on average 0.12 ms. We opt for a low latency environment so that our results better reflect the overheads due to local processing related to session types. In error-free communications ensured by session types, we expect the main extra cost incurred by our runtime to be its local processing, even with high latency.

Results In Figure 6(1), we measure the performance of STNIO against the plain NIO socket, and STSocket against the plain socket (we only show the case of the small message; the results of the large message experiment exhibit the same pattern). We observe that overheads are almost negligible for STNIO at all session lengths, but are relatively more prominent for STSocket. For the latter, our investigation [4] shows that this overhead is mostly due to the handshake at session initialisation (*not* its type validation process), which can be mitigated by an optimisation discussed later. STSocket's less efficient method to cache sent messages may also be contributing to the overhead, in the light of a gradual slowdown in longer sessions.

The next two graphs compare both session-typed sockets to RMI, in Figure 6(2a) for the small message and in Figure 6(2b) for the large message. Observe that, regardless of message size and iteration, STSocket and STNIO consistently outperform RMI. The figures also indicate that the

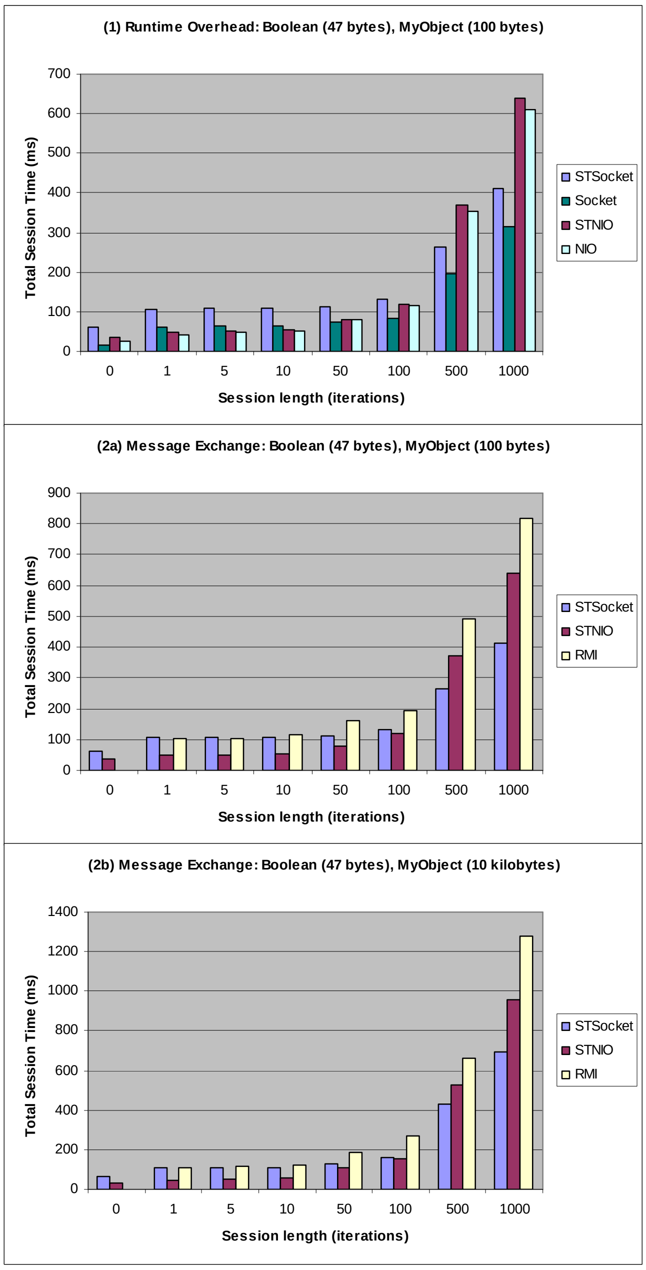


Figure 6.1. Communication performance.

cost of session initialisation is relatively cheap with respect to the overheads of the RMI protocol.

Note also STSocket is less competitive in fewer iterations, but begins to outperform STNIO after approximately 50 iterations. Since graph(1) exhibits the same trend between plain socket and plain NIO (suggesting the former has a more costly set-up but less overhead for communications with message sizes considered), we judge that this trend directly reflects that of the plain socket and the plain NIO.

We also list the mean time needed for a single iteration as part of large iterations, comparing RMI, STSocket, STNIO,

	RMI	STSocket	STNIO	Socket	NIO
100B	0.82	0.41	0.64	0.31	0.60
10KB	1.30	0.70	0.96	0.60	0.90

Table 2. Mean Time (ms) for Single Iteration out of 1000.

plain socket and plain NIO in Table 6. The figures reflect running overhead of communication actions relatively accurately since a large iteration tends to amortise the initial handshake cost. They confirm our observations on a relatively low overhead needed for a runtime for sessions, especially in the case of STNIO.

Finally, apart from benchmark results, a relative strength in performance of session-based programs may often come from its representation of communication structures. Consider the following communication pattern:

```
begin.?[!(Integer).!(String)]*.end
```

Such a structure can be directly programmed as a session, with each iteration consisting of two asynchronous sending actions and one reception. However, a program using, say, RMI may represent each iteration with three remote calls (hence three round trips). We may mitigate the overhead using "futures" [29] or by spawning threads, which however may not result in the most natural program structures.

Summary The performance results indicate that session-typed communication can be implemented with little overhead over a given transport; and that our initial, unoptimised implementations consistently outperform RMI. The merit of RMI is the closeness of the abstraction to standard method invocation: for programs with more complex conversation patterns, these performance results indicate the practical feasibility of our programming methodology.

Optimisations One of the basic sources of overhead in our implementations observed in the benchmarks is the handshake performed at session initiation. We can mitigate this problem with the information of session types. For example, if the session has the form `begin.?(String).end`, it suffices to send this type together with the value itself without the handshake. Indeed, the initial handshake can almost always be piggy-backed, making the session-based socket further competitive with respect to other means.

As another example, two parties can negotiate the method of transport based on session structures. Consider:
`begin.![!(String).!(String).!(String)]*.end`
This session type suggests a potential for message batching: the two parties may indeed agree that three messages and a control message will be delivered in one chunk at each iteration, leading to a better usage of bandwidth. The session types would be used for such negotiation scenarios which may as well differ depending on applications domains. Further investigations into how session type information, including message size and communication direction, may be exploited for efficiency concerns would be worthwhile.

7. Extensions and Related Work

7.1 Extension to Code Mobility

Higher-order code mobility allows programs to exchange higher-order code, i.e. a code which can be instantiated with objects and other higher-order code, and is useful for diverse applications, from refined applets to agent programming to optimisations. On the basis of our preceding work [2, 6], we have implemented the higher-order code mobility as part of the session-based socket. In comparison with [2, 6], we strengthened type safety, making the best of the dynamic session validation discussed in §3.

The extension offers general primitives for code mobility through a primitive for *freezing* a statement block, essentially creating a closure, and one for *defrosting*, at which point any parameters specified by the closure must be supplied and the evaluation result is returned.

```
(int,int->boolean) equals =
  freeze(int x, int y) {
    return x == y;
  };
...defrost(equals, 1, 2); // false
```

The type system checks for correct argument types and handling of the return type. In [2, 6], all frozen code types are translated to a base type of `Object[]->Object`. As a consequence, casting frozen code types is not safe in the general case. For example, in

```
...((int,int->boolean))ois.readObject();
```

where `ois` is an instance of `ObjectInputStream` wrapped around a socket, in the implementation in [2, 6], we were forced to allow the cast to succeed regardless of the actual type of the frozen code being received; incorrect types will only be caught if and when the frozen code is defrosted. With *session types*, we can assert at the receiver that `?((int,int->boolean))....`, to which the sender must adhere in order for the two parties to start communication. This example demonstrates the effective interplay between compile-time semantic verification and runtime validation, one of the key characteristics of this work. This instance also suggests a potential to exploit the validation framework for refined properties such as e.g. access control and secrecy.

7.2 Related Work

In the following we discuss those works which are directly related with the present work, with an emphasis on (the design of) concurrent languages with a theoretical basis.

Theories of Session Types Session types were first proposed in [38] as one of type disciplines for mobile processes [30]. [26] extends this work to the higher-order session passing (delegation), and [23] to the subtyping for branching and selection. Both of them form a theoretical basis of the present work. A recent article [44] analyses the type soundness of the session types in details, comparing different reduction rules and typing systems appeared in the literature [8, 23, 26, 40].

In the context of theories of session types for object-oriented languages, [17] first studied the integration of session types in a distributed Java without higher-order session passing. [16] investigated the session types for a multi-threaded Java including session passing, and proved type soundness and progress. A more recent work [14] studied an asynchronous session typing system and its progress property. The operational semantics in [14] corresponds to the implementation of this paper. [15] further extends [16] to the bounded polymorphism based on [21]. These works show that a consistent integration of session types in object-oriented languages is possible at a theoretical level. On the basis of theories of sessions developed in the preceding studies (in particular [14–17]), the present work demonstrates the use of session types as a key element of concrete programming methodology, with a concrete implementation in one of the major object-oriented languages. It shows how the incorporation of session types to a real-world language leads to a new programming discipline, a clean implementation of communication mechanism offering a compatibility check at session initiation and a transparent session delegation, and a positive impact on performance, all of which crucially using type information.

Language Design for Session Types (1) One of the initial usage of session types in practice is found in Web Services. Because of the need of static validation of safety of business protocols, a web description language called Web Service Description Languages (WS-CDL), developed by a W3C standardisation working group [37], uses a variant of session types. Execution of the web description in WS-CDL is implemented through communication among distributed end-points written in languages such as Java or WS4-BPEL [5]. The recent work [11, 12, 27, 43] studied the principles to obtain a sound and complete implementation from the web description written in CDL into the end-point processes. We plan to use the current implementation as a theoretically founded end-point implementation for WS-CDL.

An implementation of session types in Haskell is studied in [31], where the authors encode a simple calculus based on session types into Haskell. A merit of this approach is that we obtain a type checking for session types as part of 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.

Language Design for Session Types (2) In [19], a variant of the session types has been unified into a derivative of C♯ for systems programming, playing a crucial role for the development of Singularity OS in shared memory uni/multiprocessor environments. Their aim is to describe interactions among OS-modules as message passing conversations following session types: thus session types are used as contracts among modules (they indeed call their types as such). To mitigate the often unbearable cost of message passing in shared memory environments, they map message passing into linear pointer passing among threads, passing the ownership of a datum in a specifically delineated exchange area. Because of a fixed session structure, in-session communications can almost always be performed as direct pointer rewriting in a sequential thread, making their message passing faster than standard inter-process communication. Reflecting their hardware and software assumption (i.e. shared memory and homogeneity of OS modules), the initial handshake in a session-based communication is not significant (in fact substantially non-existent) in their design, unlike ours: in particular, their type discipline lacks session subtyping, which, as we discussed in §3, plays a fundamental role for practical use of session types in open environments.

Our design idea is based on a decoupling of session abstraction and its implementation, meditating two by type information. This framework can be effectively used for realising delegation of sessions, which abstracts away from physical connection, and efficient communication. [19] takes a different approach, as embodied in their use of shared region for communicable data for kernel programming. In spite of significant differences in design directions, the two works together show that using a session-based abstraction for structured, type-safe communication in objected-oriented languages have non-trivial impacts in abstraction, implementation mechanisms, and performance.

Language Design based on Process Theories The present work shares with many recent works its direction towards well-structured communication-centred programming using types. Pict [32] is the programming language based on the π-calculus, with rich type disciplines including linear and polymorphic types. Polyphonic C♯ [7] is designed based on Join-calculus and uses a type discipline for safe and sophisticated object synchronisation. The Concurrency and Coordination Runtime [1] is a port-based concurrency library for C♯ for component-based programming, whose design is based on Poly♯. JavaSeal [42] is an implementation of the Seal calculus for Java and is realised as an API and run-time system inside the JVM, targeted as a programming framework for building multi-agent systems. None of these works implemented a typing discipline which can guarantee communication safety of a sequence of interactions.

Occam-pi [3] is a highly efficient concurrent language based on channel-based communication. It is based on both Hoare's CSP (and its practical embodiment, Occam) and the π-calculus. It is designed for systems-level programming, allowing generation of more than million threads for a single processor machine without efficiency degradation. The communication model of occam-pi is point-to-point and synchronous, and allows consistent code mobility during execution. Occam-pi can realise various locking and barrier abstractions built from its highly efficient communication primitives. Typing of a larger unit than an individual communication may not have been considered.

X10 [13] is a typed concurrent language based on Java, with a quite different orientation than the present work. X10 is designed for high-performance computing. As such, one of the target environments of X10 is (possibly distributed) shared memory symmetric multi-processors. A semantic basis of X10 is studied in [35]. 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. The interplay between X10's design elements and session types is an interesting future topic.

The present work uses Java NIO package [28]. NIO allows operations on low-level events and data structures. On the one hand, the use of NIO has a potential to enable highly efficient operations on communication channels [34]. On the other, operations exported by NIO are generally error prone to program and code using it tends to get highly involved. In this context, session types offer a medium to exploit the raw interface offered by NIO package while exposing only high-level abstraction to programmers, offering opportunities for various forms of runtime tuning.

8. Conclusion and Further Topics

In this paper we presented the design and implementation of session types in Java. Session types can be used to declare a variety of structured interactions (such as business protocols between multiple distributed parties). Programming with session types is similar to socket programming, but is more structured and amenable to compositional, static type-checking. We extended Java with session primitives and the APIs for session communication, and implemented the type-checking algorithm with subtyping. We also designed and implemented session delegation using session information. Lastly, we demonstrated that our implementation framework, which depends crucially on session type information, allows typed and safe communication with minimal overhead on the underlying transports, leading to a significant performance improvement over RMI, the standard typed interface for inter-process communication.

Sessions and session types offer an abstraction layer for structured communication sequences. Our runtime decouples underlying communication mechanisms from this layer, with a mediation by the 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. The guarantee of safety of a session by the protocol hinges on two assumptions: that involved programs are statically verified with respect to declared session types, and that they communicate their session types as well as the result of compatibility validation honestly. Either maliciously or by error, if these assumptions are not met, safety is lost. Elementary measures to foil possible attacks include rejection of wrongly typed messages (which is already realised by insertion of casting) and timeout mechanisms against prolonged interactions. An in-depth study of security in session types is a future topic. Investigation of how other languages such as [1, 3, 13] may or may not benefit from session-like constructs 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 [36], making the most of session types. Other interests include a development of comprehensive programming methodology based on session types; and refined specifications along the line of e.g. [11, 14, 15]. An immediate use of the current implementation is planned for a theoretically founded automatic type-checking and code generation using a Web description language [43] and the theory in [11].

A. Full Programs in Section 2.

This appendix lists the whole code in §2, omitting the exceptions and "main" methods.

References

[1] CCR: An Asynchronous Messaging Library for C#2.0. `http://channel9.msdn.com/wiki/default.aspx/Channel9.ConcurrencyRuntime`.

[2] DJ SourceForge Homepage. `http://dj-project.sourceforge.net/`.

[3] occam-pi home page. `http://www.cs.kent.ac.uk/projects/ofa/kroc/`.

[4] On-line Appendix of this paper. `http://www.doc.ic.ac.uk/~rh105/sessiondj.html`.

[5] WS-BPEL OASIS Web Services Business Process Execution Language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-specification-draft.html.

[6] A. Ahern and N. Yoshida. Formalising Java RMI with Explicit Code Mobility. In R. Johnson and R. P. Gabriel, editors, *OOPSLA '05*, pages 403–422. ACM Press, 2005. A full version: *Journal of Theoretical Computer Science*.

[7] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for c#. *ACM Trans. Program. Lang. Syst.*, 26(5):769–804, 2004.

[8] E. Bonelli, A. Compagnoni, and E. Gunter. Correspondence Assertions for Process Synchronization in Concurrent Communications. *JFP*, 15(2):219–248, 2005.

[9] E. Bonelli, A. Compagnoni, and E. Gunter. Typechecking Safe Process Synchronization. In J. Rathke, editor, *FGUC 2004*, volume 138 of *ENTCS*, pages 3–22. Elsevier, 2005.

[10] M. Carbone, K. Honda, and N. Yoshida. A Calculus of Global Interaction Based on Session Types. In *DMC'06*, ENTCS. Elsevier, 2006.

```
private String TRAVEL_METHOD = "...";
private Double MAX_PRICE = new Double(...);
private Address ADDRESS = ...;

Customer(String host, int port) {
  STSocket agency = STSocketImpl.create(customerProtocol);
  boolean decided = false;
  int retry = 100;

  agency.request(host, port);
  agency.outwhile(!decided && (retry-- > 0)) {
    agency.send(TRAVEL_METHOD);
    if(agency.receive().compareTo(MAX_PRICE) < 0) decided = true;
  }
  if(retry > 0) {
    agency.select(ACCEPT) {
      agency.send(ADDRESS);
      System.out.println(agency.receive());
      agency.close();
    }
  }
  else {
    agency.select(REJECT);
    agency.close();
  }
}
```

```
Agency(int agencyPort, String host, int port) {
  STServerSocket agency = STServerSocketImpl.create(agencyProtocol, agencyPort);
  while(true) {
    STSocket client = agency.accept();
    client.inwhile() { client.send(getPrice(client.receive())); }
    client.branch() {
      case ACCEPT: {
        STSocket delegate = STSocketImpl.create(delegateProtocol);
        delegate.send(client);
        delegate.close();
      }
      case REJECT: {client.close();}
    }
  }
}
```

```
Service(int port) {
  STServerSocket service = STServerSocketImpl.create(delegatorProtocol, port);
  while(true) {
    STSocket delegator = service.accept();
    STSocket customer = delegator.receive();
    delegator.close();
    storeInDatabase(customer.receive());
    customer.send(getDate(...));
    customer.close();
  }
}
```

Figure A.1. Business protocol example

[11] M. Carbone, K. Honda, and N. Yoshida. Structured Communication-Centred Programming for Web Services. In *ESOP'07*, LNCS. Springer, 2007.

[12] M. Carbone, K. Honda, N. Yoshida, R. Milner, G. Brown, and S. Ross-Talbot. A theoretical basis of communication-centred concurrent programming. To be published by W3C. Available at `www.dcs.qmul.ac.uk/~carbonem/cdlpaper`.

[13] 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.

[14] M. Coppo, M. Dezani-Ciancaglini, and N. Yoshida. Asynchronous Session Types and Progress for Object-Oriented Languages. To appear in FMOOSE'07. `http://www.di.unito.it/~dezani/papers/cdy.pdf`, 2007.

[15] M. Dezani-Ciancaglini, S. Drossopoulou, E. Giachino, and N. Yoshida. Bounded Session Types for Object-Oriented Languages. Submitted for publication. `http://www.di.unito.it/~dezani/papers/ddgy.pdf`, 2007.

[16] M. Dezani-Ciancaglini, D. Mostrous, N. Yoshida, and S. Drossopoulou. Session Types for Object-Oriented Languages. In D. Thomas, editor, *ECOOP'06*, volume 4067 of *LNCS*, pages 328–352. Springer-Verlag, 2006.

[17] M. Dezani-Ciancaglini, N. Yoshida, A. Ahern, and S. Drossopoulou. A Distributed Object Oriented Language with Session Types. In R. D. Nicola and D. Sangiorgi, editors, *TGC'05*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005.

[18] 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 W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[19] 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 W. Zwaenepoel, editor, *EuroSys2006*, ACM SIGOPS, pages 177–190. ACM Press, 2006.

[20] P. Garralda, A. Compagnoni, and M. Dezani-Ciancaglini. BASS: Boxed Ambients with Safe Sessions. In M. Maher, editor, *PPDP'06*, pages 61–72. ACM Press, 2006.

[21] S. Gay. Bounded polymorphism in session types. *MSCS*. To appear.

[22] S. Gay and M. Hole. Types and Subtypes for Client-Server Interactions. In S. D. Swierstra, editor, *ESOP'99*, volume 1576 of *LNCS*, pages 74–90. Springer-Verlag, 1999.

[23] S. Gay and M. Hole. Subtyping for Session Types in the Pi-Calculus. *Acta Informatica*, 42(2/3):191–225, 2005.

[24] S. Gay, V. T. Vasconcelos, and A. Ravara. Session Types for Inter-Process Communication. TR 2003–133, Department of Computing, University of Glasgow, 2003.

[25] K. Honda. Types for Dyadic Interaction. In E. Best, editor, *CONCUR'93*, volume 715 of *LNCS*, pages 509–523. Springer-Verlag, 1993.

[26] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In C. Hankin, editor, *ESOP'98*, volume 1381 of *LNCS*, pages 22–138. Springer-Verlag, 1998.

[27] K. Honda, N. Yoshida, and M. Carbone. Web services, mobile processes and types. *EATCS*, 2, Feb 2007. To appear.

[28] D. Lea. *Scalable IO in Java*. `http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf`, November 2003.

[29] S. Microsystems Inc. New IO APIs. `http://java.sun.com/j2se/1.4.2/docs/guide/nio/index.html`.

[30] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes, Parts I and II. *Info.& Comp.*, 100(1), 1992.

[31] M. Neubauer and P. Thiemann. An Implementation of Session Types. In *PADL*, volume 3057 of *LNCS*, pages 56–70. Springer, 2004.

[32] 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.

[33] Polyglot Home Page. `http://www.cs.cornell.edu/Projects/polyglot/`.

[34] W. Pugh and J. Spacco. Mpjava: High-performance message passing in java using java.nio. In *LCPC*, pages 323–339, 2003.

[35] V. A. Saraswat and R. Jagadeesan. Concurrent clustered programming. In *CONCUR*, volume 3653 of *LNCS*, pages 353–367. Springer, 2005.

[36] A. C. Snoeren and H. Balakrishnan. An end-to-end approach to host mobility. In *MOBICOM*, pages 155–166, 2000.

[37] S. Sparkes. Conversation with Steve Ross-Talbot. *ACM Queue*, 4(2), 2006.

[38] K. Takeuchi, K. Honda, and M. Kubo. An Interaction-based Language and its Typing System. In C. Halatsis, D. Maritsas, G. Philokyprou, and S. Theodoridis, editors, *PARLE'94*, volume 817 of *LNCS*, pages 398–413. Springer-Verlag, 1994.

[39] A. Vallecillo, V. T. Vasconcelos, and A. Ravara. Typing the Behavior of Objects and Components using Session Types. In A. Brogi and J.-M. Jacquet, editors, *FOCLASA'02*, volume 68(3) of *ENTCS*, pages 439–456. Elsevier, 2002.

[40] V. T. Vasconcelos, S. Gay, and A. Ravara. Typechecking a Multithreaded Functional Language with Session Types. *Theoretical Computer Science*, 2006. To appear.

[41] V. T. Vasconcelos, A. Ravara, and S. Gay. Session Types for Functional Multithreading. In P. Gardner and N. Yoshida, editors, *CONCUR'04*, volume 3170 of *LNCS*, pages 497–511. Springer-Verlag, 2004.

[42] J. Vitek, C. Bryce, and W. Binder. Designing JavaSeal or how to make Java safe for agents. *Electronic Commerce Objects*, 1998.

[43] Web Services Choreography Working Group. Web Services Choreography Description Language. `http://www.w3.org/2002/ws/chor/`.

[44] N. Yoshida and V. T. Vasconcelos. Language Primitives and Type Disciplines for Structured Communication-based Programming Revisit. In *SecRet'06*, ENTCS. Elsevier, 2007.