

13

Distributed Programming Using Java APIs Generated from Session Types

Raymond Hu

Imperial College London, UK

Abstract

This is a tutorial on using Scribble [9], a toolchain based on multiparty session types [1, 4], for distributed programming in Java. The methodology is based on the generation of *protocol-specific Endpoint APIs* from Scribble specifications [6]. We start with a brief recap of TCP network programming using standard Java APIs, and their limitations with regards to safety assurances. The main tutorial content is an overview of the key stages of the Scribble toolchain, from global protocol specification, through Endpoint API generation, to Java endpoint implementation, with examples. We discuss the hybrid form of session safety promoted by the Endpoint API generation approach. We then consider Scribble specifications and implementations of HTTP as a real-world use case. Finally, we demonstrate some further Scribble features that leverage Endpoint API generation to safely support more advanced communication patterns.

13.1 Background: Distributed Programming in Java

The two core facilities for TCP-based network programming in Java (and other languages) are the *socket APIs* and Java *Remote Method Invocation* (RMI). The socket APIs allow the programmer to work directly with TCP connections, and are the basis over which many higher-level networking facilities are built. Java RMI is the Java adaptation of remote procedure call (RPC) functionality; with regards to this discussion, RMI is representative

of the corresponding facilities in other languages or platform-independent frameworks, *e.g.*, RESTful Web services.

Running example: Math Service. As a running Hello World example, we specify and implement a two-party network service for basic arithmetic operations. For the purposes of this tutorial, we are not concerned with the most realistic development of such a service, but rather that this simple example features core constructs of protocol design, such as message sequencing, alternative cases and repeated sequences.

Figure 13.1 depicts the Math Service protocol as a UML sequence diagram [8, §17]. For some number of repetitions (`loop`), the client `c` sends to the server `s` a `Val` message with an `Integer` payload; `c` then selects between the two alternatives (`alt`), to send an `Add` or a `Mult` message carrying a second `Integer`. `s` respectively replies with a `Sum` or a `Prod` message carrying the result. Finally, `c` sends a `Bye` message, with no payload, ending the session.

13.1.1 TCP Sockets

Sockets are supported in the standard Java API by the `java.net` and `java.nio.channels` packages. Figure 13.2 gives a client implementation using Math Service for a factorial calculation via the `java.net.Socket` API. For simplicity, we assume serializable Java classes for each of the message types (*e.g.*, `Add`), with a field `val` for the `Integer` payload, and use standard Java object serialization via `java.io.ObjectOutput/InputStream`.

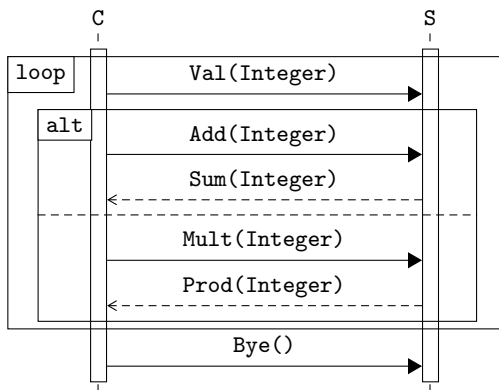


Figure 13.1 Sequence diagram for the Math Service protocol.

```

1 try (Socket s = new Socket("localhost", 8888);
2     ObjectOutputStream os = new ObjectOutputStream(s.getOutputStream());
3     ObjectInputStream is = new ObjectInputStream(s.getInputStream())) {
4     int i = 5, res = i; // Calculate 5!
5     while (i > 1) {
6         os.writeObject(new Val(i)); os.writeObject(new Add(-1)); os.flush();
7         i = ((Sum) is.readObject()).val;
8         os.writeObject(new Val(res)); os.writeObject(new Mult(i)); os.flush();
9         res = ((Prod) is.readObject()).val;
10    }
11    os.writeObject(new Bye()); os.flush();
12 }

```

Figure 13.2 A factorial calculation using Math Service via the `java.net.Socket` API.

From a networking perspective, TCP sockets offer a high-level abstraction in the sense of reliable and ordered message delivery. From an application perspective, however, the raw communication interface of a TCP channel is simply a pair of unidirectional bit streams—essentially a communications machine code, in contrast to the support for high-level data types in “local” computations.

Working directly with standard socket APIs thus affords almost no safety assurances with regards to the correctness of protocol implementations. The key kinds of application-level protocol errors are:

Communication mismatches when the sent message is not one of those expected by the receiver (also called a *reception error*). *E.g.*, if `c` were to commence a session with an `Add` message: assuming an implementation of `s` in the style of Figure 13.2, this would likely manifest as a `ClassCastException` on the object returned by the `readObject` in `s`. Note, the dual error of the receiver applying an incorrect cast are equally possible.

Deadlock in situations where some set of participants are all blocked on mutually dependent input actions. *E.g.*, if `c` were to call `readObject` after sending `Val`, but before sending `Add` or `Mult`; while `s` is (correctly) waiting for one of the latter.

Orphan messages if the receiver terminates without reading an incoming message. In practice, an orphan error often manifests as, *e.g.*, an `EOFException`, since TCP uses a termination handshake. *E.g.*, if `s` skips the receive of `Bye` before `c` has sent it, leading `c` to attempt the write on a closed connection.

13.1.2 Java RMI

RMI is a natural approach towards addressing the mismatch between high-level, typed Java programming and low-level networking interfaces. Distributed computations can be (partially) abstracted away as regular method invocations, while benefiting from static typing of each call and its communicated arguments and return value.

The Math Service protocol may be fitted to a remote interface as in Figure 13.4(a), essentially by decomposing the protocol into separate call-return fragments; and Figure 13.3 re-implements the factorial calculation as a client of this interface. Individual remote calls are now statically typed with respect to their arguments and return. Unfortunately, RMI programs in general remain subject to the same potential protocol errors illustrated for the previous sockets example (although their concrete manifestations may differ). The typed RMI interface does not prevent, for example, a bad client from calling `Add` before `Val`.

Disadvantages of call-return based protocol decomposition are further illustrated by the (minimal) implementation of the remote interface in Figure 13.4(b), which suffices to serve a single client but is completely inadequate in the presence of concurrent clients. Basic RMI programs lose the notion of an explicit *session*-oriented abstraction in the code (cf., the threading of session control flow in Figure 13.2 wrt. the socket/stream variable usages), which complicates the correlation and management of application-level session flows across the separate methods.

13.2 Scribble Endpoint API Generation: Toolchain Overview

Using the Math Service running example, we demonstrate the stages of the Scribble toolchain, from global protocol specification, through Endpoint

```
RMIMath mathS = (RMIMath) registry.lookup("MathService");
int i = 5, res = i;
while (i > 1) { mathS.Val(i);    i = mathS.Add(i - 1);
                mathS.Val(res); res = mathS.Mult(i);  }
mathS.bye();
```

Figure 13.3 Factorial calculation as a client of the remote interface in Figure 13.4(a).

```

interface RMIMath
    extends Remote {
    void Val(Integer x) .. ;
    void Bye() throws .. ;
    Integer Add(Integer y) .. ;
    Integer Mult(Integer y) .. ;
}

class RMIMathS implements RMIMath {
    private int x;
    public void Val(Integer x) ...
        { this.x = x; }
    public void Bye() throws .. { }
    public Integer Add(Integer y) ...
        { return this.x + y; }
    public Integer Mult(Integer y) ...
        { return this.x * y; }
    ...
}

```

Figure 13.4 A remote Math Service: (a) interface, and (b) implementation.

API generation, to Java endpoint implementation. The source code of the toolchain [10] and tutorial examples [5] are available online.

13.2.1 Global Protocol Specification

The tool takes as its primary input a textual description of the source protocol or choreography from a global perspective. Figure 13.5 is a Scribble *global protocol* for the Math Service running example.

A *payload format type* declaration (line 1) gives an alias (`Int`) to data type definitions from external languages (`java.lang.Integer`) used for message formatting. The *protocol signature* (line 2) declares the name of the global protocol (`MathSvc`) and the abstraction of each participant as a named *role* (`C` and `S`).

Message passing is written, *e.g.*, `Val(Int)` from `C` to `S`. A *message signature* (`Val(Int)`) declares an *operator* name (`val`) as an abstract message

```

1  type <java> "java.lang.Integer" from "rt.jar" as Int;
2  global protocol MathSvc(role C, role S) {
3    choice at C { Val(Int) from C to S;
4      choice at C { Add(Int) from C to S;
5                    Sum(Int) from S to C; }
6      or { Mult(Int) from C to S;
7            Prod(Int) from S to C; }
8      do MathSvc(C, S); }
9    or { Bye() from C to S; }
10 }

```

Figure 13.5 Scribble global protocol for Math Service in Figure 13.1.

identifier (which may be, *e.g.*, a header field value in the concrete message format), and some number of payload types (a single `Int`). Message passing is output-asynchronous: dispatching the message is non-blocking for the sender (`C`), but the message input is blocking for the receiver (`S`). A *located choice* (*e.g.*, `choice at C`) states the subject role (`C`) for which selecting one of the cases (the `or`-separated blocks) to follow is a mutually exclusive *internal* choice. This decision is an *external* choice to all other roles involved in each block, and must be appropriately coordinated by explicit messages. A `do` statement enacts the specified (sub)protocol, including recursive definitions (*e.g.*, line 8).

The body of the `MathSvc` protocol may be equivalently written in a similar syntax to standard recursive session types:

```
rec X { choice at C { Val(Int) from C to S; ... continue X; }
        or { Bye() from C to S; } }
```

Protocol validation. The tool validates the well-formedness of global protocols. We do not discuss the details of this topic in this tutorial, but summarise a few elements. Firstly, the source protocol is subject to a range of syntactic checks. Besides basic details, such as bound role names and recursion variables, the key conditions are *role enabling*, *consistent external choice subjects* and *reachability*. Role enabling is a consistency check on the (transitive) propagation of choice messages originating from a choice subject to the other roles involved in the choice. The following is a very simple example of bad role enabling:

```
choice at C { Val(Int) from S to C; ... } or { Bye() from C to S; }
```

Since the choice is `at C`, `S` should not perform any output before it is *enabled*, *i.e.*, by receiving a message that directs it into the correct choice case. The second of the above conditions requires that every message of an external choice be communicated *from* the same role.

Reachability of protocol states is imposed on a per-role basis, *i.e.*, on projections; Scribble protocols are also checked to be tail recursive per role. These rule out some basic syntactic inconsistencies (*e.g.*, sequential composition after a non-terminating recursion), and support the later FSM translation step (see below).

Together, the syntactic conditions support the main validation step based on explicit checking of safety errors (and progress violations), such as reception errors and deadlocks (outlined in § 13.1, on a bounded model of the

protocol. For example, (wrongly) replacing `Bye` by `val` will be found by the explicit error checking to be invalid.

`choice at C { Val(Int) from C to S; ... } or { Val(Int) from C to S; }`

The ambiguous (non-deterministic) receipt of the decision message by `s` from `c` (i.e., a message identified by `val` in both cases—Scribble does not introduce any implicit meta data or communications) may lead to various deadlock and orphan message errors, depending on the different permutations of `c` and `s` proceeding in the two cases. *E.g.*, if `c` proceeds in the right case and `s` in the left, then `s` will be stuck (in the “...”) waiting for an `Add/Mult` (or will encounter a broken connection error).

Endpoint FSM generation. The next key step is the generation of an *Endpoint Finite State Machine* (EFSM) for each role in the protocol. We use the term EFSM for the particular class of multiparty communicating FSMs given by Scribble’s syntax and validation. The construction is based on and extends the syntactic projection of *global types* to *local types* [4], followed by a translation to an EFSM, building on a correspondence between local types and communicating FSMs [2, 7]. The nodes of an EFSM represent the states in the localised view of the protocol for the target role, and the transitions are the communication actions performed by the role between states. The EFSM for every role of a valid global protocol is defined.

Figure 13.6 depicts the (dual) EFSMs for `c` and `s` in `MathSvc`. The initial states are numbered 1. The notation, *e.g.*, `S!Val(Int)` means output of message `val(Int)` to `s`; `? dually` denotes input. The recursive definition of this protocol manifests as the cycles returning to state 1.

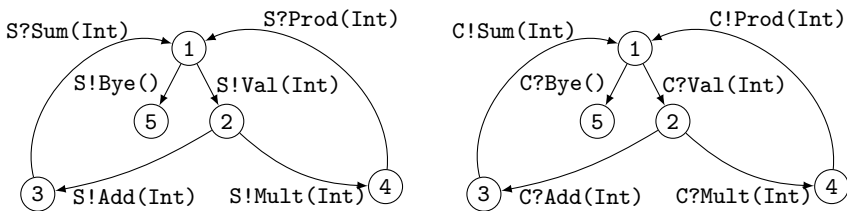


Figure 13.6 Endpoint FSMs for `C` and `S` in `MathSvc` (Figure 13.5).

13.2.2 Endpoint API Generation

For a given role of a valid global protocol, the toolchain generates an *Endpoint API* for implementing that role based on its EFSM. The current implementation generates Java APIs, but the principle may be applied or adapted to many statically-typed languages.

There are two main components of a generated Endpoint API, the *Session API* and the *State Channel API*. The generated APIs make use of a few protocol-independent base classes that are part of the Scribble runtime library: `Role`, `Op`, `Session`, `MPSTEndpoint` and `Buf`; the first three are abstract classes. These shall be explained below.

Session API. The frontend class of the Session API, which we refer to as the *Session Class*, is a generated final subclass of the base `Session` class with the same name as the source protocol, *e.g.*, `MathSvc`. It has two main purposes. One is to house the family of protocol-specific constants for type-directed session programming in Java, generated as follows.

A session type based protocol specification features various kinds of names, such as role names and message labels. A session type system typically requires these names to be present in the processes to drive the type checking (*e.g.*, [4, 1]). For the present Java setting, the Session API is generated to reify these abstract names as *singleton types* following a basic (eagerly initialised) singleton pattern. For each role or message operator name n in the source protocol, we generate:

- A final Java class named n that extends the appropriate base class (`Role` or `Op`). The n class has a single private constructor, and a public static final field of type n and with name n , initialised to a singleton instance of this class.
- In the Session Class, a public static final field of type n and with name n , initialised to the constant declared in the corresponding n class.

For example, for role `C` of `MathSvc`, the subclass `C` of `Role` is generated to declare the singleton constant `public static final C C = new C();`. The `MathSvc` class is generated to collect these various constants together, including the field `public static final C C = C.C;`.

The Session API comprises the Session Class with the singleton type classes. The other main purpose of the Session Class is for session initiation in endpoint implementations, as explained below.

An implementation of `c` via Endpoint API generation. At this point, we give, in Figure 13.7, a first version of the factorial calculation using the Endpoint API generated by the Scribble tool for `C` in `MathSvc`.

The code can be read similarly to the socket code in § 13.1.1; *e.g.*, `s1` is a session channel variable. A difference from the earlier socket code is that the Scribble API is generated as a *fluent* interface, allowing consecutive I/O operations to be chained (*e.g.*, line 11). We refer to and explain this code through the following subsections.

Session initiation. Lines 3–6 in Figure 13.7 is a typical preamble for a (client) endpoint implementation using a Scribble-generated API. We start by creating a new `MathSvc` session by instantiating the Session Class. The session object, `sess`, is used to create a new session *endpoint* object of type `MPSTEndpoint<MathSvc, C>`, parameterised on the type of the session and the endpoint role. The `C` parameter in this type is the singleton type in the Session API; and the `C` argument in the constructor call is the single value of this type.

The third argument required by the `MPSTEndpoint` constructor is an implementation of the `ScribMessageFormatter` interface, that is responsible for the underlying serialization and deserialization of individual messages in this session. For this example, we use the default `ObjectStreamFormatter` provided by the Scribble runtime library based on the standard Java serialization protocol (messages communicated by this formatter must implement the `Serializable` interface).

```

1  int facto(int n) throws Exception { // Pre: n >= 1
2    Buf<Integer> i = new Buf<>(n), res = new Buf<>(i.val);
3    MathSvc sess = new MathSvc();
4    try (MPSTEndpoint<MathSvc, C> ep =
5         new MPSTEndpoint<>(sess, C, new ObjectStreamFormatter())) {
6      ep.connect(S, SocketChannelEndpoint::new, "localhost", 8888);
7      MathSvc_C_1 s1 = new MathSvc_C_1(ep);
8      while (i.val > 1)
9        s1 = sub1(s1, i) // sub1 on line 15
10         // State transitions: 1 -> 2 -> 4 -> 1 (see Fig. 1.6)
11         .send(S, Val, res.val).send(S, Mult, i.val).receive(S, Prod, res);
12      s1.send(S, Bye); // 1 -> EndSocket
13      return res.val;
14    } }
15  MathSvc_C_1 sub1(MathSvc_C_1 s1, Buf<Integer> b) throws ... {
16    // State transitions: 1 -> 2 -> 3 -> 1 (see Fig. 1.6)
17    return s1.send(S, Val, b.val).send(S, Add, -1).receive(S, Sum, b);
18  }

```

Figure 13.7 Factorial calculation using the Endpoint API generated for `C`.

Before proceeding to the main body of a protocol implementation, the `MPSTEndpoint` object is used to set up the session topology via connection establishment actions with the appropriate peer endpoints. On line 6, the `MPSTEndpoint` is used to perform the client-side `connect` to `S`. The second argument is a reference to the constructor of `SocketChannelEndpoint` in the Scribble runtime library, which embodies a standard TCP socket; alternatives include HTTP and shared memory endpoints. The connection setup phase is concluded when the `MPSTEndpoint` is passed as a constructor argument to the initial state channel constructor, `MathSvc_C_1`, explained next.

The `MPSTEndpoint` implements the Java `AutoCloseable` interface and should be handled using a try-with-resources, as on line 4; the encapsulated network resources are implicitly closed when control flow exits the try statement.

State Channel API. The State Channel API is generated to capture the protocol-specific behaviour of a role in the source global protocol, as represented by its EFSM, via the static typing facilities of Java.

- Each state in the EFSM is reified as a Java class for a *state-specific* session channel, thus conferring a distinct Java type to channels at each state in a protocol. We refer to instances of these generated channel classes as *state channels*.
- The I/O operations (methods) supported by a channel class are the transitions permitted by the corresponding EFSM state.
- The return type of each generated I/O operation is the channel type for the successor state following the corresponding transition from the current state. Performing an I/O operation on a state channel returns a new instance of the successor channel type.

By default, the API generation uses a simple state enumeration (*e.g.*, Figure 13.6) for the generated channel class names; *e.g.*, `MathSvc_C_1` for the initial state channel. More meaningful names for states may be specified by the user as annotations in the Scribble source. The terminal state of an EFSM, if any, is generated as an `EndSocket` class that supports no further I/O operations. The channel class for the initial state is the only class with a public constructor, taking an `MPSTEndpoint` parameterised on the appropriate Session Class and role types; all other state channels are instantiated internally by the generated API operations.

Figure 13.8 summarises the generated channel classes and their main I/O operations for `C` in `MathSvc`. *E.g.*, a state channel of type `MathSvc_C_1` supports methods for sending `Val` and `Bye` to `S`; these `send` methods are overloaded via

Gen. class	Session operation methods
MathSvc_C_1	MathSvc_C_2 <code>send(S role, Val op, Integer pay1)</code> EndSocket <code>send(S role, Bye op)</code>
MathSvc_C_2	MathSvc_C_3 <code>send(S role, Add op, Integer pay1)</code> MathSvc_C_4 <code>send(S role, Mult op, Integer pay1)</code>
MathSvc_C_3	MathSvc_C_1 <code>receive(S role, Sum op, Buf<? super Integer> pay1)</code>
MathSvc_C_4	MathSvc_C_1 <code>receive(S role, Prod op, Buf<? super Integer> pay1)</code>

Figure 13.8 State Channel API generated for C in MathSvc (Figure 13.5).

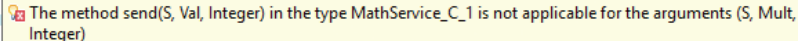
the parameters for the destination role and message operator (the singleton types of the Session API), as well as the message payloads. Sending a `Val` returns a `MathSvc.C_2` channel, *i.e.*, the state of sending an `Add` or `Mult`; whereas a sending a `Bye` returns an `EndSocket`.

For unary input states, *i.e.*, an EFSM state with a single input transition, the generated `receive` method, *e.g.*, for `Sum` in `MathSvc.C_3`, takes `Buf` arguments parameterised according to the expected payload type(s), if any. `Buf<T>` is a simple generic one-slot buffer provided by the Scribble runtime library, whose value is held in a public `val` field. The `receive` method is generated to write the payload(s) of the received message to the respective `Buf` arguments.

In Figure 13.7, lines 7–14 use the State Channel API for `c` to perform the factorial calculation. Starting from the instance of `MathSvc.C_1`, assigned to `s1`, the implementation proceeds by performing *one* I/O operation on each current state channel to obtain the next. The fluent API permits convenient chaining of I/O operations, *e.g.*, line 17 in `sub1` starts from state 1, and proceeds through states 2 and 3 (by sending `Val` and `Add` messages), before returning to 1 (by receiving the `Sum`). The endpoint implementation is complete upon reaching `EndSocket`.

Attempting any I/O action that is not permitted by the current protocol state, as designated by the target state channel, will be caught by Java type checking. For example (from the Eclipse IDE):

```
s1 = sub1(s1, i)//.send(S, Val, b.val) -- did not send the Val
    .send(S, Mult, i.val).receive(S, Prod, b);
```

 The method `send(S, Val, Integer)` in the type `MathService_C_1` is not applicable for the arguments `(S, Mult, Integer)`

13.2.3 Hybrid Session Verification

As demonstrated above, Scribble-generated Endpoint APIs leverage standard, static Java type checking to verify protocol conformance, *provided every state channel returned by an API operation is used exactly once up to the end of*

the session. This is the implicit usage contract of a Scribble-generated API, to respect EFSM semantics in terms of following state transitions linearly up to the terminal state.

Much research has been conducted towards static analyses for such resource usage properties: to this end, it may be possible to combine these with API generation tools to recover fully static safety guarantees in certain contexts. However, designing such analyses for mainstream engineering languages, such as Java and C#, in full generality is a significant challenge, and often based on additional language extensions or imposing various conservative restrictions.

As a practical compromise, the Endpoint API generation of Scribble promotes a *hybrid* approach to session verification. The idea is simply to complement the static type checking of session I/O on state channels with *run-time* checks that each state channel is indeed used exactly once in a session execution.

Run-time checking of linear state channel usage. The checks on linear state channel usage are inlined into the State Channel API operations by the API generation. There are two cases by which state channel linearity may be violated.

Repeat use. Every state channel instance maintains a boolean state value indicating whether it has been *used*, *i.e.*, a session I/O operation has been performed on the channel. The API generation guards each I/O operation with a run-time check on this boolean. If the channel has already been used, a `LinearityException` is raised.

Unused. All state channels of a session instance share a boolean state value indicating whether the session is *complete* for the local endpoint. The API is generated to set this flag when a *terminal operation*, *i.e.* an I/O action leading to the terminal EFSM state, is performed. If control flow leaves the enclosing `try` statement of the associated `MPSTEndpoint`, the Scribble runtime checks this flag via the implicit `close` method of the `AutoCloseable` interface. If the session is incomplete, an exception is raised.

It is not possible for the completion flag to be set if any state channel remains unused on leaving the `try` statement of an `MPSTEndpoint`. IDEs

(e.g., Eclipse) support compile-time warnings in certain situations where `AutoCloseable` resources are not appropriately handled by a `try`.

Hybrid session safety. Together, a statically typed Endpoint API with runtime state channel linearity checking offers the following properties.

1. If a session endpoint implementation respects state channel linearity, then the generated API statically ensures freedom from the application errors outlined in § 13.1 (i.e., *communication safety*, e.g., [4, error-freedom]) when composed with conformant endpoints for the other roles in the protocol.
2. Regardless of state channel linearity, any statically well-typed endpoint implementation will never perform a message passing action that does not conform to the protocol.

These properties follow from the fact that the only way to violate the EFSM of the API, generated from a validated protocol, is to violate state channel linearity, in which case the API raises an exception *without* actually performing the offending I/O action. This hybrid form of session verification thus guarantees the absence of protocol violation errors during session execution up to premature termination, which is always a possibility in practice due to program errors outside of the immediate session code, or other failures, such as broken connections.

When following the endpoint implementation pattern promoted by a generated API, by associating session code to the `MPSTEndpoint-try`, the Java `IOException` of, e.g., a broken connection will direct control flow out of the `try`, safely (w.r.t. session typing) avoiding further I/O actions in the failed session. Finer-grained treatment of session failures is a direction of ongoing development for Scribble (and MPST).

13.2.4 Additional Math Service Endpoint Examples

A first implementation of `s`. Figure 13.9 summarises the State Channel API generated for `s` in `MathSvc`. Unlike `c`, the EFSM for `s` features non-unary input states, which correspond at the process implementation level to the *branch* primitive of formal session calculi (e.g., [1]). Java does not directly support a corresponding language construct, but API generation enables some different options.

One option, demonstrated here, is designed for standard Java `switch` patterns. For each branch state, a *branch-specific* enum is generated to enumerate

Generated class	Session operation methods
MathSvc_S_1	MathSvc_S_1_Cases <code>branch(C role)</code>
MathSvc_S_1_Cases	MathSvc_S_2 <code>receive(Val op, Buf<? super Integer> pay1)</code> EndSocket <code>receive(Bye op)</code>
MathSvc_S_2	MathSvc_S_2_Cases <code>branch(C role)</code>
MathSvc_S_2_Cases	MathSvc_S_3 <code>receive(Add op, Buf<? super Integer> pay1)</code> MathSvc_S_4 <code>receive(Mult op, Buf<? super Integer> pay1)</code>
MathSvc_S_3	MathSvc_S_1 <code>send(C role, Sum op, Integer pay1)</code>
MathSvc_S_4	MathSvc_S_1 <code>send(C role, Prod op, Integer pay1)</code>

```

1  try (ScribServerSocket ss = new SocketChannelServer(8888)) { // TCP
2    while (true) { // Persistent server
3      MathSvc sess = new MathSvc();
4      try (MPSTEndpoint<MathSvc, S> ep
5          = new MPSTEndpoint<>(sess, S, new ObjectOutputStreamFormatter())) {
6        ep.accept(ss, C);
7        Buf<Integer> b1 = new Buf<>(), b2 = new Buf<>();
8        MathSvc_S_1 s1 = new MathSvc_S_1(ep);
9        Loop: while (true) {
10       MathSvc_S_1_Cases c1 = s1.branch(C);
11       switch (c1.op) {
12         case Bye: c1.receive(Bye); break Loop;
13         case Val:
14           MathSvc_S_2_Cases c2 = c1.receive(Val, b1).branch(C);
15           switch (c2.op) {
16             case Add: s1 = c2.receive(Add, b2)
17                       .send(C, Sum, b1.val + b2.val); break;
18             case Mult: s1 = c2.receive(Mult, b2)
19                       .send(C, Prod, b1.val * b2.val); break;
20           }
21       }
22     }
23   }

```

Figure 13.9 State Channel API generated for *S* in *MathSvc*; and an implementation of *S* using the generated API.

the cases of the choice according to the source protocol. *E.g.*, for the initial state of *S*: `enum MathSvc_S_1_Enum { Val, Bye }`.

The channel class itself (Figure 13.9), *MathSvc_S_1*, is generated with a single `branch` operation. This method blocks until a message is received, returning a new instance of the generated *MathSvc_S_1_Cases* class, which holds the enum value corresponding to the received message in a final `op` field. Unfortunately, since the *static* type of the *Cases* object reflects the *range* of possible cases, the API requires the user to manually call the corresponding `receive` method of the *Cases* object, essentially as a form of cast to obtain the appropriately typed state channel.

Lines 11–20 in Figure 13.9 implement a `switch` on the `op` enum of `MathSvc_S_1_Cases`. The Java compiler is able to statically determine whether all enum cases are exhaustively handled. In each of the two cases (`Bye` and `Val`), the corresponding `receive-cast` is called on the `Cases` object to obtain the successor state channel of that (input) transition. Leveraging the hybrid verification approach, the generated API includes an implicit run-time check that the correct cast method is used following a branch; calling an incorrect method raises an exception.

§ 13.4 discusses an alternative API generation that allows session branches to be checked by Java typing *without* additional run-time checks.

Alternative c factorial implementation. Following is an implementation of a factorial calculation using the `C` endpoint of `MathSvc` in a recursive method, illustrating the use of the State Channel API in an alternative programming style.

```
MathSvc_C_1 facto(MathSvc_C_1 s1, Buf<Integer> b) throws ... {
    if (b.val == 1) return s1; // Pre: b.val >= 1
    Buf<Integer> tmp = new Buf<>(b.val);
    return facto(sub1(s1, tmp), tmp) // sub1 from Fig. 13.7
        .send(S, Val, b.val).send(S, Mult, tmp.val).receive(S, Prod, b);
}
```

Besides conformance to the protocol itself, the state channel parameter and return types help to ensure that the appropriate I/O transitions are performed through the protocol states in order to enact the recursive method call correctly.

13.3 Real-World Case Study: HTTP (GET)

In this section, we apply the Scribble API generation methodology to a real-world protocol, HTTP/1.1 [3]. For the purposes of this tutorial, we limit this case study to the GET method of HTTP, and treat a minimal number of message fields required for interoperability with existing real-world clients and servers. The following implementations have been tested against Apache (as currently deployed by the dept. of computing, Imperial College London) and Firefox 5.0.1.

A key point illustrated by this experiment on using session types in practice is the interplay between *data types* (message structure) and *session types* (interaction structure) in a complete protocol specification. In particular, that

aspects of the former can be refactored into the latter, while fully preserving protocol interoperability, to take advantage of the safety properties offered by Scribble-generated APIs in endpoint implementations.

13.3.1 HTTP in Scribble: First Version

HTTP is well-known as a client-server request-response protocol, typically conducted over TCP. Despite its superficial simplicity, *i.e.*, a sequential exchange of just two messages between two parties, the standards documentation for HTTP spans several hundred pages, as is often the case for Internet applications and other real-world protocols.

Global protocol. As a first version, we simply express the high-level notion of an HTTP request-response as follows:

```
sig <java> "...client.Req" from ".../Req.java" as Req;
sig <java> "...server.Resp" from ".../Resp.java" as Resp;
global protocol Http(role C, role S) {
  Req from C to S;
  Resp from S to C;
}
```

A small difference from the Scribble examples seen so far are the `sig` declarations for custom message formatting. Unlike `type` declarations, which pertain specifically to payload types, `sig` is used to work with host language-specific (*e.g.*, `<java>`) routines for arbitrary message formatting; *e.g.*, `Req.java` contains Java routines, provided as part of this protocol specification, for performing the serialization and deserialization between Java `Req` objects and the actual ASCII strings that constitute concrete HTTP requests on the wire.

Client implementation. For such a simple specification, we omit the EFSMs and Endpoint APIs for each role, and directly give client code using the generated API (omitting the usual preamble):

```
Buf<Resp> b = new Buf<>();
Http_C_1 s1 = new Http_C_1(client); // client: MPSTEndpoint<Http, C>
s1.send(S, new Req("/index.html", "1.1", host)).receive(S, Resp, b);
```

The generated API prevents errors such as attempting to receive `Resp` before sending `Req` or sending multiple `Reqs`. However, one may naturally wonder if this is “all there is” to a correct HTTP client implementation—where is the complexity that is carefully detailed in the RFC specification?

The answer lies in the message formatting code that we have conveniently abstracted as the `Req` and `Res` message classes. A basic HTTP session does exchange only two messages, but these messages are richly structured, involving branching, optional and recursive structures. In short, this first version *assumes* the correctness of the `Req` and `Res` classes (written by the protocol author, or obtained using other parsing/formatting utilities) as part of the protocol specification.

13.3.2 HTTP in Scribble: Revised

As defined in RFC 7230 [3] (§ 3 onwards), the message grammar is:

```
HTTP-message = start-line *( header-field CRLF ) CRLF [ message-body ]
start-line   = request-line / status-line
request-line = method SP request-target SP HTTP-version CRLF
header-field = field-name ":" OWS field-value OWS
... // CRLF=carriage return line feed; SP=space; OWS=optional white space
```

Intuitively, the act of sending a HTTP request may be equivalently understood as sending a request-line, followed by sending zero or more header-fields terminated by CRLF, and so on. Following this intuition, we can refactor much of this structure from the data side of the specification to the session types side, giving a Scribble description that captures the target protocol specification in more explicit detail than previously. Consequently, the generated API will promote the Java endpoint to respect this finer-grained protocol structure by static typing, as opposed to assuming the correctness of the supplied message classes.

We are able to refine the Scribble for HTTP/TCP in this way because any application-level notion of “message” identified in the specification is ultimately broken down and communicated via the TCP bit streams, in a manner that is transparent to the other party (client or server). This approach may thus be leveraged for any application protocol conducted over a transport with such characteristics.

Global protocol. Figure 13.10 is an extract of a revised Scribble specification of HTTP. The monolithic request and response messages have been decomposed into smaller constituents; *e.g.*, `RequestL` and `Host` respectively denote the request-line and host-field in a request. For the most part, the Java code for formatting each message fragment as an HTTP ASCII string is reduced to a simple print instruction with compliant white spacing built in

```

1 sig <java> "...client.RequestLine" from ".../RequestLine.java" as RequestL;
2 sig <java> "...client.Host" from ".../Host.java" as Host;
3 sig <java> "...Body" from ".../Body.java" as Body;
4 sig <java> "...client.UserAgent" from ".../UserAgent.java" as UserA;
5 sig <java> "...server.HttpVersion" from ".../HttpVersion.java" as HttpV;
6 sig <java> "...server._200" from ".../_200.java" as 200;
7 sig <java> "...server._404" from ".../_404.java" as 404;
8 ...
9 global protocol Http(role C, role S) {
10   do Request(C, S);
11   do Response(C, S);
12 }
13 aux global protocol Request(role C, role S) {
14   RequestL from C to S; // GET /index.html HTTP/1.1
15   rec X {
16     choice at C { Host from C to S; continue X; } // host: www.doc.ic.ac.uk
17       or { UserA from C to S; continue X; } // User-Agent: Mozilla..
18       or ...
19       or { Body from C to S; }
20 } // (aux bypasses validating these "subprotocols" as "root" protos)
21 aux global protocol Response(role C, role S) {
22   HttpV from S to C; // HTTP/1.1
23   choice at S { 200 from S to C; } // 200 OK
24     or { 404 from S to C; } // 404 Not Found
25     ...
26 }

```

Figure 13.10 Extract from the revised specification of HTTP in Scribble.

(e.g, CRLFs). The structure by which these constituents should be composed to reform whole messages is now expressed in the Request and Response subprotocols.

Client implementation. Taking the revised Scribble HTTP, Endpoint API generation proceeds as usual, generating the EFSMs for each role to give the structure of the State Channel API. Lines 2–3 in Figure 13.11 is an almost minimal implementation of a *correctly formatted* request according to the Request subprotocol. The typed API ensures the initial, mandatory RequestLine; then amongst the recursive choice cases we opt to send only the Host field, before concluding the request by an empty Body. A complete client implementation is given by: doResponse(doRequest(s1)).

Besides limiting to a subset of the protocol, this revision is by no means a complete specification of HTTP in terms of capturing the entire message grammar in full detail; the fidelity of the Scribble specification may be pushed

```

1  Http_C_3 doRequest(Http_C_1 s1) throws Exception {
2      return s1.send(S, new RequestLine("/index.html", "1.1"))
3          .send(S, new Host("www.host.com")).send(S, new Body(""));
4  }
5  EndSocket doResponse(Http_C_3 s3) throws Exception {
6      Http_C_4_Cases cases = s3.async(S, HttpV, new Buf<>()).branch(S);
7      switch (cases.op) {
8          case _200: ...
9          case _404: ...
10         ...
11     }
12     ...

```

Figure 13.11 Extract from an implementation of a HTTP client via API generation.

further, perhaps towards a “character-perfect” specification, via suitably fine-grained message decomposition.

13.4 Further Endpoint API Generation Features

Branch-specific callback interfaces. Scribble also generates a *callback-based API* for branch states, which does *not* require additional run-time checks (cf. § 13.2.4). For each branch state, a *handler* interface is generated with a callback variant of `receive` for each choice case; e.g., `MathSvc_S_1_Handler` in Figure 13.12. Apart from the operator and payloads, each method takes the continuation *state channel* as a parameter; the return type is `void`. Java typing ensures that a (concrete) implementation of this interface implicitly covers all cases of the branch. Finally, a variant of `branch` is generated in the parent channel class (e.g., `MathSvc_S_1`) that takes an instance of the corresponding handler interface, with return `void`. As before, this `branch` blocks until a message is received; the API then delegates the handling of the message to the appropriate callback method of the supplied handler object.

Figure 13.12 gives a class that implements the handler interfaces of *both* branch states for `s`. Assuming an `MPSTEndpoint<MathSvc, S> serv`, this handler class may be used in an event-driven implementation of `s` by: `new MathSvc_S_1(serv).branch(C, new MathSHandler())`.

State-specific futures for unary inputs. For unary input states, Scribble additionally generates *state-specific input futures* as an alternative mechanism

Generated class	Session operation methods (additional to Fig. 1.9)
MathSvc_S_1	void branch (C role, MathSvc_S_1_Handler h)
MathSvc_S_1_Handler	void receive (MathSvc_S_2 s, Val op, Buf<Integer> pay1) void receive (EndSocket s, Bye op)
MathSvc_S_2	void branch (C role, MathSvc_S_2_Handler h)
MathSvc_S_2_Handler	void receive (MathSvc_S_3 s, Add op, Buf<Integer> pay1) void receive (MathSvc_S_4 s, Mult op, Buf<Integer> pay1)
1	class MathSHandler implements MathSvc_S_1_Handler, MathSvc_S_2_Handler {
2	private Buf<Integer> b1;
3	public void receive (MathSvc_S_2 s2, Val op, Buf<Integer> b1) .. {
4	this .b1 = b1;
5	s2. branch (C, this);
6	}
7	public void receive (EndSocket end, Bye op) throws ... { }
8	public void receive (MathSvc_S_3 s3, Add op, Buf<Integer> b2) throws ..
9	{ s3. send (C, Sum, this .b1.val + b2.val). branch (C, this); }
10	public void receive (MathSvc_S_4 s4, Mult op, Buf<Integer> b2) throws ..
11	{ s4. send (C, Prod, this .b1.val * b2.val). branch (C, this); }
12	}

Figure 13.12 Additional branch callback interfaces generated for S in MathSvc; and a corresponding implementation of S.

to the basic `receive`. For example, in the revised Scribble specification of HTTP (Figure 13.11), the channel class `Http_C_3` corresponds to the state where `c` should receive the HTTP version element (`HttpV`) of the response status-line (Line 14 in Figure 13.10). For this state, Scribble generates the class `Http_C_3_Future`. Its key elements are input-specific fields for the message type (`msg`) or payloads (e.g., `pay1`) to be received, and a `sync` method to force the future. For the `Http_C_3` channel class itself, the following variant of `receive` is generated:

```
Http_C_4 async(S role, HttpV op, Buf<Http_C_3_Future> fut)
```

Unlike a basic `receive`, calling `async` returns immediately with a new instance of `Http_C_3_Future` in the supplied `Buf`.

Calling `sync` first implicitly forces all pending prior futures, in order, for the same peer role. It then blocks the caller until the expected message is received, and writes the values to the generated fields of the future. This safely preserves the FIFO messaging semantics between each pair of roles in a session, so that endpoint implementations using generated futures retain the same safety properties as using only blocking receives. Repeat forcing of an input future has no effect.

An example usage of `async` was given in Figure 13.11 (line 6). There, the `async` is used to safely affect a *non-blocking input* action (the client is

not interested in blocking on awaiting just the `HTTPV` portion of the response). Since the `HTTPV` future is never explicitly forced – unlike state channels, input-futures are not linear objects – `async` also affects a *user-level* form of *affine* input action, in the sense that the user never reads this message. Finally, `async` enables postponing input actions until later in a session, for safe user-level *permutation* of session I/O actions.

References

- [1] M. Coppo, M. Dezani-Ciancaglini, N. Yoshida, and L. Padovani. Global progress for dynamically interleaved multiparty sessions. *Mathematical Structures in Computer Science*, 760:1–65, 2015.
- [2] P.-M. Deniérou and N. Yoshida. Multiparty session types meet communicating automata. In *ESOP '12*, volume 7211 of *LNCS*, pages 194–213. Springer, 2012.
- [3] R. Fielding, Y. Lafon, M. Nottingham, and J. Reschke. IETF RFCs 7230–7235 Hypertext Transfer Protocol 1.1. <https://tools.ietf.org/html/rfc7230>
- [4] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL '08*, pages 273–284. ACM, 2008.
- [5] R. Hu. Demo files for this BETTY tutorial chapter. <https://github.com/scribble/scribble-java/tree/master/modules/demos/scrib/bettybook>
- [6] R. Hu and N. Yoshida. Hybrid session verification through endpoint API generation. In *FASE '16*, volume 9633 of *LNCS*, pages 401–418. Springer, 2016.
- [7] J. Lange, E. Tuosto, and N. Yoshida. From communicating machines to graphical choreographies. In *POPL '15*, pages 221–232. ACM Press, 2015.
- [8] OMG UML 2.5 specification. <http://www.omg.org/spec/UML/2.5>
- [9] Scribble homepage. <http://www.scribble.org>
- [10] Scribble GitHub repository. <https://github.com/scribble/scribble-java>

