Session Scala

Multiparty Session Programming
with Scala, Scribble and AMQP

http://code.google.com/p/session-scala/

Olivier Pernet — Imperial College London
Lisbon Behavioural Types Workshop
April 19th, 2011

This talk

- Intro to Session Scala and Scribble
- Session type + implementation examples
- Session initiation in Session Scala
- Ongoing work:
    - One-to-many choice
    - Nested role invitations

Session Scala

- No new syntax, only extra compile-time checks
- Runtime based on / interop. with Scala Actors library
- Implemented as a Scala compiler plugin + small runtime lib

```
> scalac -Xplugin:sessions.jar -cp sessions-rt.jar myfile.scala
```

  - Code will compile even without plugin
  - Currently, lib supports shared memory and AMQP communication

- Uses Scribble as the session type declaration language

Scribble

- Multiparty session type definition language
- Independent of programming languages
- Includes framework for well-formedness validation, session type projection, runtime monitoring
- Open-source: www.scribble.org
- Main developer: Gary Brown, Red Hat, with contributions from others including myself
- Usable now, more features coming

Examples

Basic example - evenserver.spr

```
protocol EvenServer(role Client, role EvenServer) {

  Int from Client to EvenServer;

  Boolean from EvenServer to Client;

}
```

Basic example - evenserver.scala

```
val client = newLocalAddress("evenserver.spr", 'Client)
val serv = newLocalAddress("evenserver.spr", 'EvenServer)
actor { startSession(client, serv) }
actor {
   client.bind { s =>
      s ! 'EvenServer -> 42
      println("is even: " + s.?[Boolean]('EvenServer))
   }
}
serv.bind { s =>
   val i = s.?[Int]('Client)
   s ! 'Client -> i % 2 == 0
}
```

Choice example - opserver.spr

```
protocol OpServer(role Client, role OpServer) {
  choice from Client to OpServer {
    add(Int, Int): Int from OpServer to Client;
    even(Int): Boolean from OpServer to Client;
    time(): Date from OpServer to Client;
    String: String from OpServer to Client;
  }
}
```

Choice example - opserver.scala

```
client.bind { s =>
  s ! ‘OpServer -> (‘add, 42, 1)
  println(“42 + 1 = ” + s.?[Int](‘OpServer))
}

opserv.bind { s =>
  s.receive(‘Client) {
    case (‘add, i: Int, j: Int) => s ! ‘Client -> i+j
    case (‘even, i: Int) => s ! ‘Client -> i % 2 == 0
    case ‘time => s ! ‘Client -> new Date
    case str: String => s ! ‘Client -> “You said: ”+str
} }
```

Recursion example - recserver.spr

```
protocol RecServer(role Client, role RecServer) {

  rec X {

     choice from Client to RecServer {

        Int:

           Boolean from RecServer to Client;

           X;

        quit():

} }
```

```
client.bind { s =>
  def loop(s: SessionChannel) {
    if (wantMore) {
      s ! ‘RecServer -> 42
      println(“is even: ” + s.?[Boolean] (‘RecServer))
      loop(s)
    } else s ! ‘RecServer -> ‘quit
  }
  loop(s)
}
recserv.bind { s =>
  def loop(s: SessionChannel) {
    s.receive(‘Client) {
      case i: Int => s ! ‘Client -> i % 2 == 0 ; loop(s)
      case ‘quit =>
    }
  }
  loop(s)
}
```

Session initiation

Session initiation

- Using `startSession` and `bind`
  - One process calls `startSession`, sends out invite messages for each role in the session (possibly to itself) to given addresses
  - Other processes block on `bind`, waiting for an invite
  - Alternatively, a process can block on `forward` and forward the invite to another
  - Can start multiple instances of a protocol by calling `startSession` again / in a loop
  - Implementation of theory work under submission by Tzu-Chun Chen et al.

Session initiation protocol

- Inviter creates unique reply address
- Inviter sends invite to destination addresses including reply address
- Invited processes either forward, or accept the invitation
    - accept: send message to reply address, including confirmation address and session address
- Inviter waits for confirmations for all invites, then sends the map role - session addresses to all confirmation addresses
- All session participants can start

Ongoing work

Problem: Servers / Services

```
protocol ClientMidServ(role Client, role Middleware, role Service) {
    request() from Client to Middleware;
    choice from Middleware to Service {
        nothing():
          simpleReply() from Middleware to Client;
        subrequest():
          subreply() from Service to Middleware;
          complexReply() from Middleware to Client;
}}
```

Solution

```
protocol ClientMidServ(role Client, role Middleware) {
    request(...) from Client to Middleware;
    choice at Middleware {
      Middleware introduces Service;
      subrequest(...) from Middleware to Service;
      complexReply(...) from Service to Client;
    } or {
      simpleReply(...) from Middleware to Client;
    }
}
```

Projection: Client

```
protocol ClientMidServ(role Middleware, role Service)@Client {
    request(...) to Middleware;
    do {
      complexReply(...) from Service;
    } or {
      simpleReply(...) from Middleware;
    }
}
```

Projection: Service

```
protocol ClientMidServ(role Client, role Middleware)@Service {
  subrequest(...) from Middleware;
  complexReply(...) to Client;
}
```

Projection: Middleware

```
protocol ClientMidServ(role Client)@Middleware {
    request(...) from Client;
    do {
      introduce Service;
      subrequest(...) to Service;
    } or {
      simpleReply(...) to Client;
    }
}
```

Implementation: Client

```
client.bind { s =>
  s ! ‘Middleware -> (‘request, ...)
  s.mreceive {
    case ‘Middleware -> (‘simpleReply, ...) =>
    case ‘Service -> (‘complexReply, ...) =>
  }
}
```

Added benefits - 1

```
protocol P(role A, role B) {
    choice at A {
      rec X {
        M1 from A to B;
        X;
      }
    } or {
      M2 from A to B;
    }
}
```

Added benefits - 2

```
protocol P(role A, role B) {
    choice at A {
      run Sub1(A,B);
    } or {
      run Sub2(A,B);
    }
}
```

Implementation: Client

```
client.bind { s =>
  s ! ‘Middleware -> (‘request, ...)
  s.mreceive {
    case ‘Middleware -> (‘simpleReply, ...) =>
    case ‘Service -> (‘complexReply, ...) =>
  }
}
```

Thanks. Questions?

Multiparty Session Programming
with Scala, Scribble and AMQP

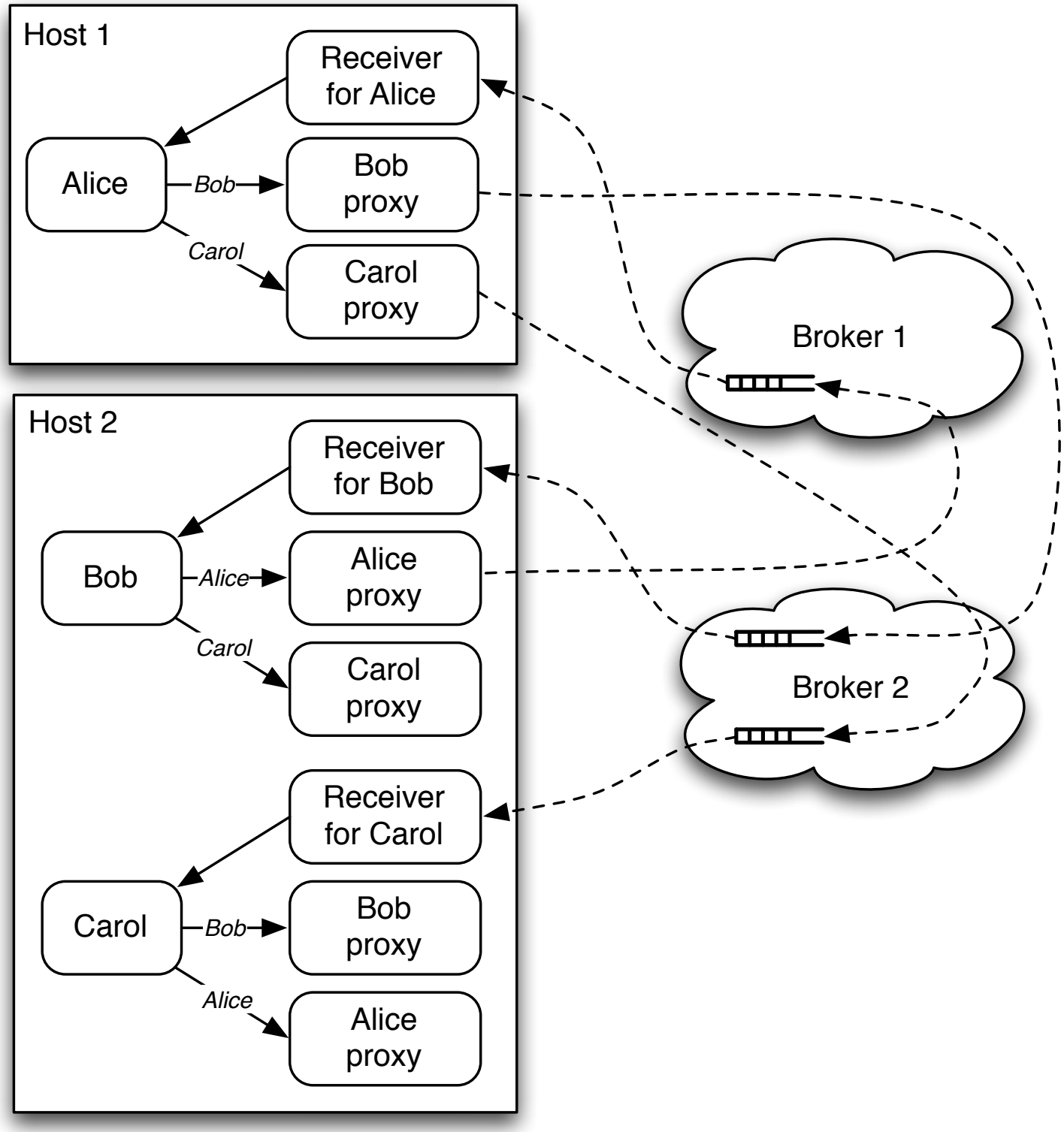
http://code.google.com/p/session-scala/

Olivier Pernet — Imperial College London
Lisbon Behavioural Types Workshop
April 19th, 2011

Backup slides

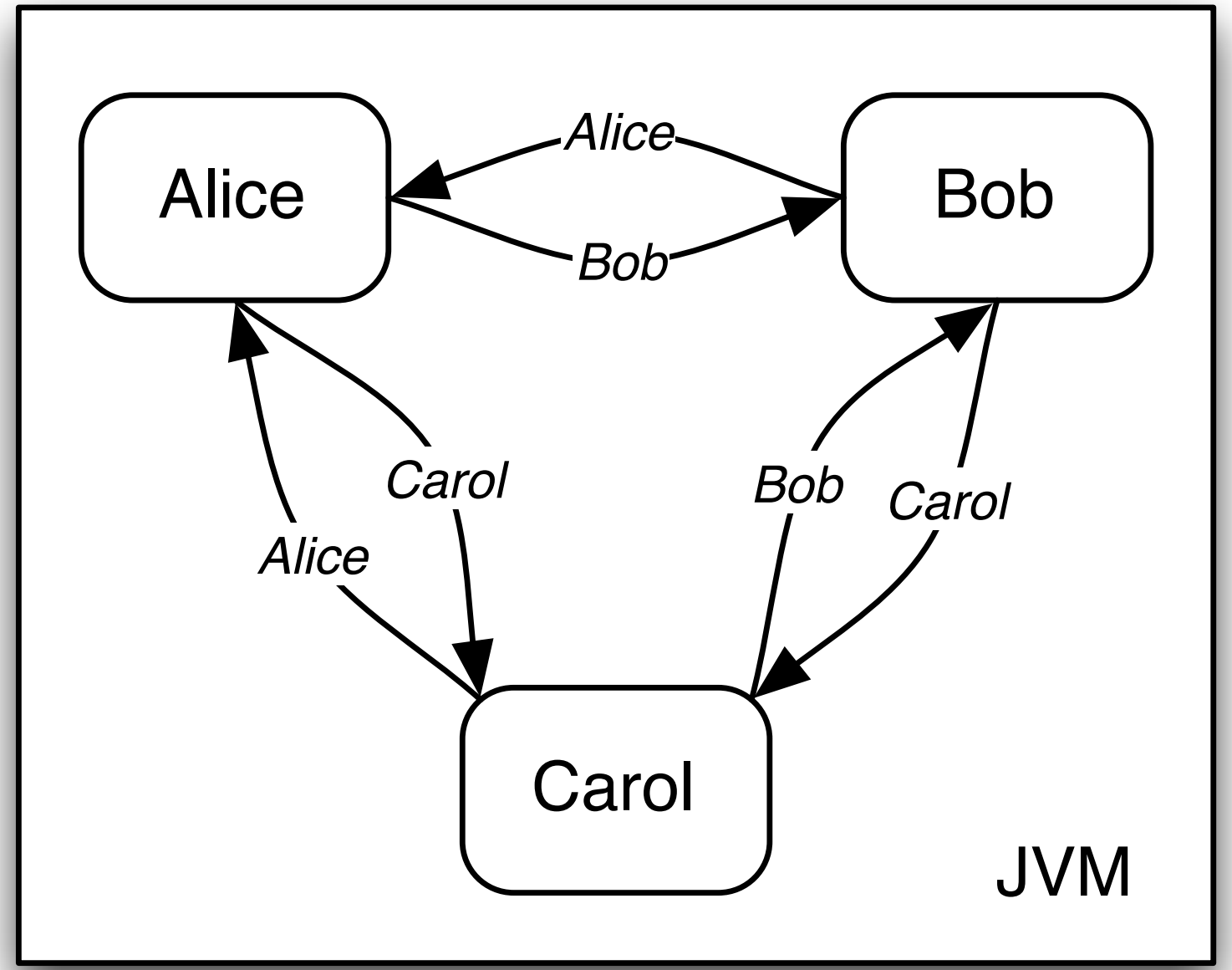
AMQP communication

- AMQP: Emerging Internet standard for async message passing
- Using very small part of AMQP
  - Compatible with current 0.9.1 and upcoming 1.0
- “Addresses” in the abstract model map to `queue@broker`
- Each `SessionChannel` (s) uses
  - an actor proxy for each role in session
  - one receiver actor, consuming messages sent to process’ session address


Host 1
Receiver for Alice
Alice
Bob
Bob proxy
Carol
Carol proxy
Broker 1
Host 2
Receiver for Bob
Bob
Alice
Alice proxy
Carol
Carol proxy
Broker 2
Receiver for Carol
Carol
Bob
Bob proxy
Alice
Alice proxy

Shared memory communication

- Using Scala actors as directly as possible
- “Addresses” in the abstract model map to a scala.actors.Channel
  - Channel: tag on message in actor mailbox
- Each `SessionChannel` (s) uses
  - A Channel for each role in the session to send to others
  - The current actor mailbox to receive messages


Alice
Bob
Carol
Alice
Bob
Alice
Carol
Bob
Carol
JVM