Multiparty Session Actors

Rumyana Neykova, Nobuko Yoshida

Multiparty Session Actors

Rumyana Neykova and Nobuko Yoshida

Imperial College London

Abstract. Actor coordination armoured with a suitable protocol description language has been a pressing problem in the actors community. We study the applicability of multiparty session type (MPST) protocols for verification of actor programs. We incorporate sessions to actors by introducing minimum additions to the model such as the notion of actor roles and protocol mailboxes. The framework uses Scribble, which is a protocol description language based on multiparty session types. Our programming model supports actor-like syntax and runtime verification mechanism guaranteeing communication safety of the participating entities. An actor can implement multiple roles in a similar way as an object can implement multiple interfaces. Multiple roles allow for cooperative interconcurrency in a single actor. We demonstrate our framework by designing and implementing a session actor library in Python and its runtime verification mechanism. Benchmark results demonstrate that the runtime checks induce negligible overhead.

1 Introduction

The actor model [2,3] is (re)gaining attention in the research community and in the mainstream programming languages as a promising concurrency paradigm. Unfortunately, the programming model itself does not ensure correct sequencing of interactions between different computational processes. A study in [26] points out that "the property of no shared space and asynchronous communication can make implementing coordination protocols harder and providing a language for coordinating protocols is needed". This reveals that the coordination of actors is a challenging problem when implementing, and especially when scaling up an actor system.

To overcome this problem, we need to solve several shortcomings existing in the actor programming models. First, although actors often have multiple states and complex policies for changing states, no general-purpose specification language is currently in use for describing actor protocols. Second, a clear guidance on actor discovery and coordination of distributed actors is missing. This leads to adhoc implementations and mixing the model with other paradigms which weaken its benefits [26]. Third, no verification mechanism (neither static nor dynamic) is proposed to ensure correct sequencing of actor interactions. Most actor implementations provide static typing within a single actor, but the communication between actors – the complex communication patterns that are most likely to deadlock – are not checked.

We tackle the aforementioned challenges by studying applicability of multyparty session types (MPST) [14], a type theory for communicating processes, to actor systems. The tenet of MPST safety assurance methodology is the use of a high-level, global

specification (also called protocol) for describing the interactions of communication entities. From the global protocol each entity is given a local session type defining the order and the payload type of the interactions. Programs are then written as a collection of possibly interleaved conversations, verified against prescribed protocols and constraints at runtime. The practical incarnation of the theoretical MPST is the protocol specification language Scribble [24]. Scribble is equipped with a verification mechanism, which makes protocol creation easier and protocol verification sound. Declarative protocol specifications in Scribble can readily avoid typical errors in communications programming, including type errors, disrespect of call orders, circular service dependencies and deadlocks. We call these safety properties ensured by MPST *communication safety*.

Recent works from [18,15] prove the suitability of Scribble and its tools for the dynamic verification of real world complex protocols [20] and present a runtime verification framework that guarantees safety and session fidelity of the underlying communications. The verification mechanism is applied to a large cyberinfrastructure. The MPST dynamic verification framework is built on a runtime layer for protocol management and developers use MPST primitives for communication, which limits the verification methodology to a specific infrastructure. In this paper, we take the MPST one step further. We adapt and distil the model to present a MPST verification of actors systems. A main departure from our previous work is that [18,15] required special conversation runtime, built into the application, which restrict the programming style and the model applicability. In this paper, we prove the generality of MPST framework by showing Scribble protocols offer a wider usage, in particular, for actor programming.

Our programming model is grounded on three new design ideas: (1) use Scribble protocols and their relation to finite state machines for specification and runtime verification of actor interactions; (2) augment actor messages and their mailboxes dynamically with protocol (role) information; and (3) propose an algorithm based on virtual routers (protocol mailboxes) for the dynamic discovery of actor mailboxes within a protocol. We implement a session actor library in Python to demonstrate the applicability of the approach. To the best of our knowledge, this is the first design and implementation of session types and their dynamic verification toolchain in an actor library.

The paper is organised as follows: § 2 gives a brief overview of the the key features of our model and presents the running example. § 3 describes the constructs of Session Actors and highlights the main design decisions. § 4 presents the implementation of the model on concrete middleware. § 5 evaluates the framework overheads, compares it with a popular Scala actor library [4] and shows applications. Finally, § 6 discusses related work and concludes. The code for the runtime and the monitor tool, example applications and omitted details are available at [25].

2 Session Actors Programming Model

2.1 Actor Models and Design Choices

Actor Model Overview We assume the following actor features to determine our design choices. Actors are concurrent autonomous entities that exchange messages asynchronously. An actor is an entity (a class in our framework) that has a mailbox and

a behaviour. Actors communicate between each other only by exchanging messages. Upon receiving a message, the behaviour of the actor is executed, upon which the actor can send a number of messages to other actors, create a number of actors or change its internal state. Each actor is equipped with a mailbox where messages are buffered. Actors have states and behaviours. Behaviours of an actor can only be changed by that actor itself, while processing a message. Active threads within actors continuously process messages whenever their mailboxes are not empty. There are only three primitive operations each actor can perform: (1) create new actors; (2) send messages to other actors whose address the sender knows; and (3) become an actor with a new state and behaviour. All these operations are atomic. The only order inherent is a causal order, and the only guarantee provided is that messages sent by actors will eventually be processed by their receivers. Other synchronisation and coordination constraints need to be externally enforced, by the unit called *roles*. The notion of roles is crucial in our framework and is explained below.

Session Actors To verify actor interactions, we introduce *multiparty roles* that enable multiple local control flows inside an actor. Each actor is annotated with supported protocols and roles it implements. As a result, session actors are containers for roles. Each actor also holds a reference to the other participating roles in the protocols. These references are bound to their physical actor containers through the actor discovery mechanism (during the protocol creation, explained in § 4.3), Without session annotations a session actor behaves identically to a plain actor. In a nutshell, an actor can be transformed to a session actor by applying the following design methodology:

1. the global protocol the actor is part of is written in Scribble and projected to local specifications;
2. the actor class is annotated with a `@protocol` decorator, which links to the local protocol specification;
3. each method is annotated with `@role` decorator that exposes a role instance (acting as a container for all protocol roles); and
4. interactions with other actors are performed via the exposed role instance.

The changes and additions that the MPST annotations bring to the original actor model are as follows (we link to the subsections where they are explained):

(a) different passive objects (called roles) that run inside an actor are given a session type (§ 4.2);
(b) an actor engages in structured protocol communications via a protocol mailbox (§ 4.3) and dynamically learns the roles it is communicating with; and
(c) an actor message execution is bound to a protocol structure (§ 4.4). This structure is checked via the internal FSM-based monitor.

This design choice (incorporating roles inside actors) enables to apply the MPST verification framework to actors, as explained in this and the next two sections.

Scribble overview Scribble [24,13] is a practical and human-readable language for protocol specification that is designed from the multiparty session types theory [14,6]. It is a language to describe application-level protocols among communicating systems and aims to prevent communication mismatches and type errors during communications. A Scribble protocol represents an agreement on how participating systems interact with each other by describing an abstract structure of message exchanges between roles.

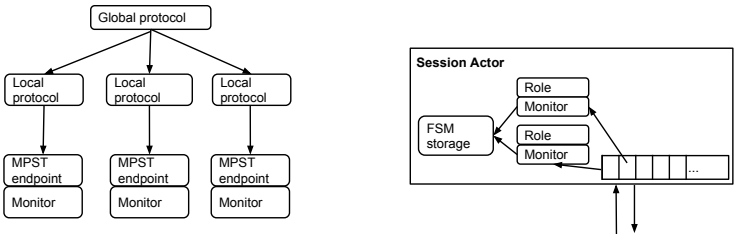


Fig. 1: MPST-development methodology (left) and session actors (right)

Roles abstract from the actual identity of the endpoints that may participate in a runtime conversation instantiating the protocol. The core structures supported in Scribble are asynchronous message passing, choice, parallel and recursion constructs. The Scribble toolkit comes with libraries for parsing, well-formedness checking algorithms and several language bindings. An implementation of Python-based Scribble tools for projection and validation [24], as well as static verification for various languages, e.g.[19] are explained in the literature [12].

MPST verification framework The top-down development methodology for MPST verification is shown in the left part of Fig. 1 and the additions for session actors is illustrated on the right.

A distributed protocol is specified as a global Scribble protocol, which collectively defines the admissible communication behaviours between the participating entities, called *roles* inside the protocol. Then, the Scribble toolchain is used to algorithmically project the global protocol to local specifications.

For each role a finite state machine (FSM), which prescribes the allowed actions and communicating parties, is generated from the local specification and stored in a distributed storage to be retrieved whenever a role is instantiated. When a party requests to start or join a session, the initial message specifies which role it intends to play as. Its monitor retrieves the local specification based on the protocol name and the role. A runtime monitor ensures that each endpoint program conforms to the core protocol structure.

To guarantee session fidelity we perform two main checks. First, we verify that the type (operation and payload) of each message matches its specification (operations can be mapped directly to message headers, or to method calls, class names or other relevant artefacts in the program). Second, we verify that the overall order of interactions is correct, i.e. interaction sequences, branches and recursions proceed as expected, respecting the explicit dependencies (for example, `m1() from A to B; m2() from B to C;` where `A`, `B` and `C` denote roles and `m1()` and `m2()` denote methods imposes an input-output causality). These measures rule out errors, e.g. communication mismatches, that violate the permitted protocol.

Verification of Session Actors We explain how we apply the above methodology. As observed in [23] in many actor systems an actor encapsulates a number of passive objects, accessed from the other actors through asynchronous method calls. Similarly session actors are a collection of interleaving control flows of actor roles. An actor role is associated with a FSM, generated from a global protocol. This verification structure is depicted in the right hand side of Fig. 1.

The association is done through annotating the actor type (class) and receive messages (methods) with protocol information. Annotating the actor type with protocol information results in registering the type for a particular role. When a session is started, a join message is sent (in a round-robin style) to all registered actors. When this join message is received, the generated FSM is loaded into the actor role and all subsequent messages on that protocol (role) are tracked and checked via the verification mechanism explained above. Message receive is delegated to the appropriate FSM via pattern matching on the protocol id, contained in the message. If all actors messages comply to their assigned FSMs, the whole communication is guaranteed to be safe. If participants do not comply, violations (such as deadlocks, communication mismatch or breakage of a protocol) are detected and delegated to a Policy actor. Further implementation details are explained in § 4.

2.2 Warehouse Management Use Case

To illustrate and motivate central design decisions of our model, we present the buyer-seller protocol from [14] and extend it to a full warehouse management scenario. A warehouse consists of multiple customers communicating to a warehouse provider. It can be involved in a `Purchase` protocol (with customers), but can also be involved in a `StoreLoad` protocol with dealers to update its storage.

Scribble Protocol. The interactions between the entities in the system are presented as two Scribble protocols, shown in Fig. 2(a) and 2(b). The protocols are called `Purchase` and `StoreLoad` and involve three (a Buyer (B), a Seller (S) and an Authenticator (A)) and two (a Store (*S*), a Dealer (D)) parties, respectively. At the start of a purchase session, B sends login details to S, which delegates the request to an Authentication server. After the authentication is completed, B sends a request quote for a product to S and S replies with the product price. Then B has a choice to ask for another product, to proceed with buying the product, or to quit. By buying a product the warehouse decreases the product amount it has in the store. Products in stock are increased as prescribed by the `StoreLoad` protocol, Fig. 2(b). The protocol starts with a recursion where the warehouse (in the role of S) has a choice to send a product request to a dealer (D) to load the store with n numbers of a product. After receiving the request, D delivers the product (operation `put` on Line 8). These interactions are repeated in a loop until S decides to `quit` the protocol (Line 11). The reader can refer to [24] for the full specification of the Scribble syntax.

Challenges. There are several challenging points to implement the above scenario. First, a warehouse implementation should be involved in both protocols, therefore it can play more than one role. Second, initially the user does not know the exact warehouse it is buying from, therefore the customer learns dynamically the address of the warehouse. Third, there can be a specific restriction on the protocol that cannot be expressed as system constants (such as specific timeout depending on the customer). The next section explains the implementation of this example in Session Actors.

```
global protocol Purchase(role B,
  role S, role A)
{
  login(string:user) from B to S;
  login(string:user) from S to A;
  authenticate(string:token) from A to B, S;
  choice at B
    {request(string:product) from B to S;
    (int:quote) from S to B;}
  or
    {buy(string:product) from B to S
     delivery(string) from S to B; }
  or
    {quit() from B to S; }}
```

```
global protocol StoreLoad
  (role D, role S)
{
  rec Rec{
  choice at S
    {request(string:product, int:n)
      from S to D;)
     put(string:product, int:n) from D to S;
     continue Rec;}
  or
    {quit() from S to D;
     acc() from D to S;}}}
```

Fig. 2: Global protocols in Scribble for (a) Purchase protocol and (b) StoreLoad protocol

3 Session Actor Language

This section explains the main operations in the session actor language and its usecase implementation in Python.

Session Actor language operations Fig. 3 presents the main session actor operations and annotations. The central concept of the session actor is the *role*. A role can be considered as a passive object inside an actor and it contains meta information used for MPST-verification. A session actor is registered for a role via the `@protocol` annotation. The annotation specifies the name of the protocol and role the session actor is registered for and the rest of the participants in the protocol. The aforementioned meta information is stored in a role instance created in the actor. To be used for communication, the method should be annotated with the role using the `@role` decorator and passing the role instance name. The instance serves as a container for references to all protocol roles, which allows sending a message to a role in the session actor without knowing the actor location. A message is sent via `c.role.method`, where `c` is the self role instance and `role` is the role the message is intended for.

Sending to a role without explicitly knowing the actor location is possible by the actor discovery mechanism (the details will be explained in § 4). When a session is started via the `create` method, actors receive an invitation for joining the protocol. The

Conversation API operation	Purpose
`@protocol(variable_name, protocol_name, self_role, other_roles) actor_class`	Annotating actor class
`@role(self_role, sender_role) msg_handler`	Annotating message handler
`c.create(protocol_name, invitation_config.yml)`	Initiating a conversation, sending invitations
`c.role.method(payload)`	Sending a message to an actor in a role
`join(self, role, principal_name)`	Actor handler for joining a protocol
`actor.send.method(payload)`	Sending a message to a known actor

Fig. 3: Session Actor operations

```
1  @protocol(c, Purchase, seller, buyer, auth)
2  @protocol(c1, StoreLoad, store, dealer)
3  class Warehouse(SessionActor):
4    @role(c, buyer)
5    def login(self, c, user):
6      c.auth.send.login(user)
7
8    @role(c, buyer)
9    def buy(self, c, product):
10     self.purchaseDB[product]-=1;
11     c.seller.send.delivery(product.details)
12     self.become(update, product)
13
14   @role(c, buyer)
15   def quit(self, c):
16     c.send.buyer.acc()
17
18   @role(c1, self)
19   def update(self, c1, product):
20     c1.dealer.send.request(product, n)
21
22   @role(c1, dealer)
23   def put(self, c1, product):
24     self.purchaseDB[product]+=1:
```

Fig. 4: Session Actor implementation for the Warehouse role

operation `join` is a default handler for invitation messages. If a session actor changes the join behaviour and applies additional security policies to the joiners, the method should be overloaded. Introducing actor roles via protocol executions is a novelty of our work: without roles and the actor discovery mechanism, actors need additional configurations to introduce their addresses, and this would grow the complexity of configurations.

Warehouse service implementation We explain the main constructs by an implementation of a session actor accountable for the Warehouse service. Fig. 4 presents the implementation of a warehouse service as a single session actor that keeps the inventory as a state (`self.purchaseDB`). Lines 1–2 annotate the session actor class with two protocol decorators – `c` and `c1` (for seller and store roles respectively). `c` and `c1` are accessible within the warehouse actor and are holders for mailboxes of the other actors, involved in the two protocols.

All message handlers are annotated with a role and for convenience are implemented as methods. For example, the login method (Line 5) is invoked when a `login` message (Line 4, Fig. 2 (a)) is sent. The role annotation for `c` (Line 4) specifies the sender to be `buyer`.

The handler body continues following Line 5, Fig. 2 (a) – sending a `login` message via the `send` primitive to the session actor, registered as a role `auth` in the protocol of `c`. Value `c.auth` is initialised with the `auth` actor mailbox as a result of the actor discovery mechanism (explained in the next section). The handling of `authenticate` (Line 6, Fig. 2 (a)) and `request` (Line 6, Fig. 2 (b)) messages is similar, so we omit it and focus on the `buy` handler (Line 9–12), where after sending the delivery details (Line 11), the warehouse actor sends a message to itself (Line 12) using the primitive `become` with value `update`. Value `update` is annotated with another role `c1`, but has as a sender `self`. This is the mechanism used for switching between roles within an actor. Update method (Line 19–20) implements the request branch (Line 6–9, Fig. 2 (b)) of the `StoreLoad` protocol – sending a request to the `dealer` and handling the reply via method `put`.

The correct order of messages is verified by the FSM attached to `c` and `c1`. As a result, errors such as calling `put` before `update` or executing two consecutive updates, will be detected as invalid.

4 Implementations of Session Actors

This section explains our implementation of Session Actors. The key design choices follow the actor framework explained in § 2.1. We have implemented the multiparty session actors on top of Celery [8] (a Python framework for distributed task processing) with support for distributed actors [1]. Celery uses advanced message queue protocol (AMQP 0-9-1 [5]) as a transport. The reason for choosing AMQP network as base for our framework is that AMQP middleware shares a similar abstraction with the actor programming model, which makes the implementation of distributed actors more natural.

4.1 AMQP Background

We first summarise the key features of the AMQP model. In AMQP, messages are published by producers to entities, called exchanges (or mailboxes). Exchanges then distribute message copies to queues using binding rules. Then AMQP brokers (virtual routers) deliver messages to consumers subscribed to queues. Exchanges can be of different types, depending on the binding rules and the routing strategies they implement. We explain the three exchange types used in our implementation: *round-robin* exchange (deliver messages, alternating to all subscribers), *direct* exchange (subscribers subscribe with a binding key and messages are delivered when a key stored in the message meta information matches the binding key of the subscription) and *broadcast* exchange (deliver a message to all subscribers).

Distributed actors are naturally represented in this AMQP context using the abstractions of exchanges. Each actor type is represented in the network as an exchange and is realised as a consumer subscribed to a queue based on a pattern matching on the actor id. Message handlers are implemented as methods on the actor class.

Our distributed actor discovery mechanism draws on the AMPQ abstractions of exchanges, queues and binding, and our extensions to the actor programming model are built using Python advanced abstraction capabilities: two main capabilities are *greenlets* (for realising the actors inter-concurrency) and *decorators* (for annotating actor types and methods).

A greenlet (or micro/green thread) is a light-weight cooperatively-scheduled execution unit in Python. A Python decorator is any callable Python object that is used to modify the function, method or class definition it annotates using the `@` symbol. A decorator is passed the original object being defined and returns a modified object, which is then bound to the name in the definition. These decorators in Python are partly inspired by Java annotations.

4.2 Actor Roles

A key idea of actor roles is each role to run as a micro-thread in an actor (using Python greenlet library). Actors are assigned to session roles by adding the `@protocol` decorator to the actor class declaration. Methods that implement a part of a protocol are annotated with the `@role` decorator. A role is activated when a message is received and ready to be processed. Switching between roles is done via the `become` primitive (as demonstrated

in Fig. 4), which is realised as sending a message to the internal queue of the actor. Roles are scheduled cooperatively. This means that at most one role can be active in a session actor at a time.

4.3 Actors Discovery

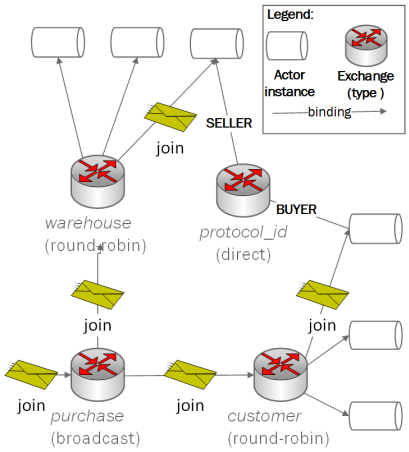


Fig. 5: Organising Session Actors into protocols

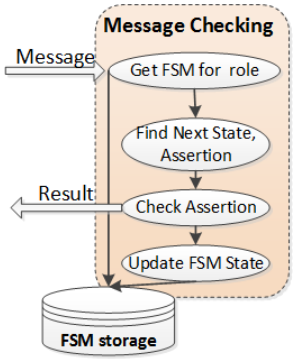


Fig. 6: Session Actors Monitoring

Fig. 5 presents the network setting (in terms of AMQP objects) for realising the actor discovery for `buyer` and `seller` of the protocol `Purchase`. We use the three types from AMQP explained in § 4.1. For simplicity, we create some of the objects on starting of the actor system – round-robin exchange per actor type (`warehouse` and `customer` in Fig. 5) and broadcast exchange per protocol type (`purchase` in Fig. 5). All spawned actors alternate to receive messages addressed to their type exchange. Session actors are registered for roles via the protocol decorator and as a result their type exchange is bound to the protocol exchange (Line 1 in Fig. 4 binds `warehouse` to `purchase` in Fig. 5).

We now explain the workflow for actor discovery. When a protocol is started, a fresh protocol id and an exchange with that id are created. The type of the exchange uses AMPQ type, direct explained in § 4.1. A direct type is used so that messages with a routing key are delivered to actors linked to the exchange with binding to that key (it corresponds to `protocol id` in Fig. 5). Then `join` message is sent to the protocol exchange and delivered to one actor per registered role (`join` is broadcasted to `warehouse` and `customer` in Fig. 5). On `join`, an actor binds itself to the `protocol id` exchange with subscription key equal to its role (bindings `seller` and `buyer` in Fig. 5). When an actor sends a message to another actor within the same session (for example `c.buyer.send` in Fig. 4), the message is sent to the protocol id exchange (stored in `c`) and from there delivered to the `buyer` actor.

4.4 Preservation through FSM checking

Before a message is dispatched to its message handler, the message goes through a monitor. Fig. 6 illustrates the monitoring process. Each message contains meta information (a routing key) with the role name the message is intended for and the id of the protocol the message is in the part of. When an actor joins a protocol, the FSM, generated from the Scribble compiler (as shown in Fig. 1) is loaded from a distributed storage to the actor memory.

Then the checking goes through the following steps. First, depending on the role and the protocol id the matching FSM is retrieved from the actor memory. Next the FSM checks the message labels/operators (already in the part of the actor payload) and sender and receiver roles (in the part of the message binding key implemented as our extension) are valid.

The *check assertions* step verifies that if any constraints on the size/value of the payload are specified in Scribble, they are also fulfilled. If a message is detected as wrong the session actor throws a distributed exception and sends an error message back to the sending role and does not pass the message to its handler for processing. This behaviour can change by implementing the `wrong_message` method of the session actor.

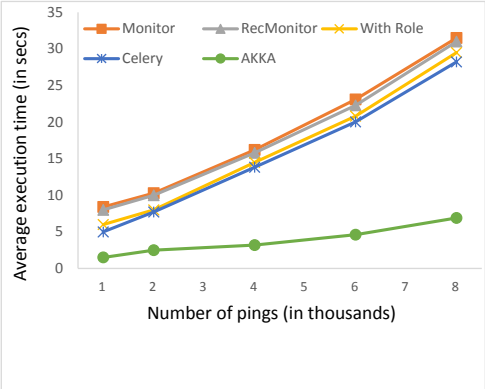
5 Evaluations of Session Actors

This section reports on the performance of our framework. The goal of our evaluation is two fold. First, we compare our host distributed actor framework [8] with a mainstream actor library (AKKA [4]) to show our host framework is a suitable choice. Second, we show that our main contribution, verification of MPST protocols, can be realised with reasonable cost. The full source code of the benchmark protocols and applications and the raw data are available from the project page [25].

5.1 Session Actors Performance

We test the overhead in message delivery implementation using the pingpong benchmark [16] in which two processes send each other messages back and forth. The original version of the code was obtained from Scala `pingpong.scala` from `http://scala-lang.org/old/node/54` and adapted to use distributed AKKA actors (instead of local). We distinguish two protocols. Each pingpong can be a separate session (protocol `FlatPingPong`) or the whole iteration can be part of one recursive protocol (`RecPingPong`). The protocols are given in Fig. 7(b). This distinction is important only to session actors, because the protocol shape has implications on checking. For AKKA actors the notion of session does not exist and therefore the two protocols have the same implementation.

Set Up We prepared each scenario 50 times and measured the overall execution time (the difference between successive runs was found to be negligible). The creation and population of the network was not measured as part of the execution time. The client and server actor nodes and the AMQP broker were each run on separate machines (Intel Core2 Duo 2.80 GHz, 4 GB memory, 64-bit Ubuntu 11.04, kernel 2.6.38). All hosts are




```
global protocol FlatPingPong(
  role C, role S)
{
  ping(string) from C to S;
  pong(string) from S to C;
}


global protocol RecPingPong(
  role C, role S)
{
  rec Loop{
    ping(string) from C to S;
    pong(string) from S to C;
    continue Loop
  }
}
```

Fig. 7: (a) The PingPong benchmark (the overhead of message delivery) and (b) The PingPong protocol

interconnected through Gigabit Ethernet and Latency between each node was measured to be 0.24 ms on average (ping 64 bytes). The version of Python used was 2.7, of Scala – 2.10.2 and of the AKKA-actors – 2.2.1.

Results Fig. 7(a) compares three separate benchmark measurements for session actors. The base case for comparison. ”Celery” is a pure actor implementation without the addition of roles. ”With Role” measures the overhead of roles annotations without having the monitor capabilities enabled. The two main cases, ”Rec Monitor” and ”Monitor”, measure the full overhead of session actors. This separation aims to clearly illustrate the overhead introduced by each of the additions, presented in the paper: roles annotations, MPST verification and actor discovery. Note that the FSMs for the recursive and for the flat protocol have the same number of states. Therefore, the observed difference in the performance is a result of the cost of the actor discovery.

The goal of the benchmarks is two fold. First, to compare the actor framework that we use with one of the most widely used frameworks for distributed actors. Second, to evaluate the performance implications of our extensions. A difference between the performance of AKKA and Celery is not surprising and can be explained with a distinct nature of the frameworks. AKKA runs on top of JVM and the remote actors in these benchmarks use direct TCP connections. On the other hand, Celery is Python-based framework which uses a message middleware as a transport, which adds additional layer of indirection. Given these differences, Celery actors have reasonable performance and are a viable alternative.

Regarding our additions we can draw several positive conclusions from the benchmarks: (1) the cost of the FSM checking is negligible and largely overshadowed by the cost of the communication (such as latency and routing); and (2) the cost of the actor discovery is reasonable, given the protocol load.

States	100	1000	10000
Time (ms)	0.0479	0.0501	0.0569

Table 1: Execution Time for checking protocol with increasing length

5.2 MPST Verification Overhead

In this subsection, we discuss the row overhead of monitor checking, which is a main factor that can affect the performance and scalability of our implementation. Knowing the complexity of checking protocols of different length is useful to reason on the applicability of the model.

We have applied two optimisations to the monitor, presented in [18,15], which significantly reduce the runtime overhead. First, we pre-generate the FSM based on the global protocol. Second, we cache the FSM protocols the first time when a role is loaded. Therefore, the slight runtime overhead, observed in the previous section is a result of the time required to choose the correct FSM among all in-memory FSMs inside the actor and the time required to check the FSM transition table that the message is expected. Both have the linear complexity on the number of roles running inside the actor and the number of states inside the FSM respectively. Table 1 shows the approximate execution time for checking a protocols of increasing length. The small numbers are omitted since the time taken is negligible.

5.3 Applications of Session Actors

As a practical evaluation of our framework, we have implemented two popular actor applications and adapted them to session actors.

The first application is a distributed chat with multiple chat rooms, where a number of clients connect to a reactive server and execute operations based on their privileges. Depending on privileges some clients might have extended number of allowed messages. We impose this restriction dynamically by annotating the restricted operations with different roles. An operation is allowed only if its actor has already joined a session in the annotated role.

The second usecase is a general `MapReduce` for counting words in a document. It is an adaptation from an example presented in the official website for celery actors: `http://cell.readthedocs.org/en/latest/getting-started/index.html`. The session actor usage removes the requirement for the manual actor spawning of `Reducers` actors inside the `Mapper` actor, which reduces the code size and the complexity of the implementation.

We give in Fig. 8 the implementation of the latter example. For sake of space and clarity, the implementation reported here abstracts from technical details that are not important for the scope of this paper–interested readers can find the full sources in [25]. The protocol is started on Line 26 specifying the type of the actors for each role and passing the arguments for the initial execution – `n` (the number of reducers) and `file` (the name of the source file to be counted). The `Mapper` implements the `join` method, which is invoked when the actor joins a protocol. Since the `Mapper` is the starting role,

```
#======SCRIBBLE CODE======
global protocol WordCount(
  A, R[n], M):
  rec Loop{
    count_lines(string) from M to R[1..n];
    aggregate(string) from R[1..n] to A;
    continue Loop;}
#======PYTHON CODE======
@protocol(c, WordCount, R, A, M)
class Mapper(Actor):
  # invoked on protocol start
  @role(c)
  def join(self, c, file, n):
    self.count_document(self, file, n)

  @role(c, self)
  def count_document(self, c, file, n):
      with open(file) as f:
          lines = f.readlines()
            count = 0
            for line in lines:
              reducer = c.R[count % n]
              count+=1
                reducer.count_lines(line)
```

```
#start the protocol
Protocol.create(WordCount,
  M=Mapper, R=Reducer, A=Aggregator,
  n=10, file="file1.txt")

@protocol(c, WordCount, A, M, R)
class Aggregator(Actor):
  @role(c, master)
  def aggregate(self, c, words):
      for word, n in words.iteritems():
          self.result.setdefault(word, 0)
            self.result[word] += n
      # when a treshhold is reached
      # print the results
      c.close()

@protocol(c, WordCount, R, A, M)
class Reducer(Actor):
  @role(c, M)
  def count_lines(self, c, line):
      words = {}
        for word in line.split(" "):
            words.setdefault(word, 0)
            words[word] += 1
            c.A.aggregate(words)
```

Fig. 8: WordCount in Session Actors

when it joins, it sends a message to itself, scheduling the execution of `count_document`. Note that spawning and linking of the actors are not specified in the code because the actor discovery mechanism accounts for it. The protocol proceeds by `Mapper` sending one message for each line of the document. The receiver of each message is one of the `Reducers`. Each `Reducer` counts the words in its line and sends the result to the `Aggregator` (`c.A`, Line 48), which stores all words in a dictionary and aggregates them. When a threshold for the result is reached, the `Aggregator` prints the result and stops the session explicitly.

Our experiences with session actors are promising. Although they introduce new notions, i.e. a protocol and a role, we found that their addition to a class-based actor framework is natural to integrate without requiring a radical new way of thinking. The protocol and role annotations are matched well with typical actor applications, and even they result in simplifying the code by removing the boilerplate for actor discovery and coordination. The protocol-oriented approach to actor programming accounts for the early error detection in the application design and the coordination of actors. The runtime verification guarantees and enforces the correct order of interactions, which is normally ensured only by hours of testing.

6 Related Work

Behavioural and session types for actors and objects There are several theoretical works that have studied the behavioural types for verifying actors [17,9]. The work [9] proposes a behavioural typing system for an actor calculus where a type describes a sequence of inputs and outputs performed by the actor body. In [17], a concurrent

fragment of Erlang is enriched with sessions and session types. Messages are linked to a session via correlation sets (explicit identifiers, contained in the message), and clients create the required references and send them in the service invocation message. The developed typing system guarantees that all within-session messages have a chance of being received. The formalism is based on only binary sessions.

Several recent papers have combined session types, as specification of protocols on communicating channels, with object-oriented paradigm. A work close to ours is [10], where a session channel is stored as a field of an object, therefore channels can be accessed from different methods. They explicitly specify the (partial) type for each method bound to a channel. Our implementation also allows modularised sessions based on channel-based communication. Their work [10] is mainly theoretical and gives a static typing system based on binary session types. Our work aims to provide a design and implementation of runtime verification based on multiparty session types, and is integrated with existing development frameworks based on Celery [8] and AMQP [5].

The work in [7] formalises behaviours of non-uniform active objects where the set of available methods may change dynamically. It uses the approach based on spatial logic for a fine grained access control of resources. Method availability in [7] depends on the state of the object in a similar way as ours.

See [27] for more general comparisons between session types and other frameworks in object-oriented paradigms.

Other Actor frameworks The most popular actor's library (the AKKA framework [4] in Scala studied in [26]) supports FSM verification mechanism (through inheritance) and typed channels. Their channel typing is simple so that it cannot capture structures of communications such as sequencing, branching or recursions. These structures ensured by session types are the key element for guaranteeing deadlock freedom between multiple actors. In addition, in [4], channels and FSMs are unrelated and cannot be intermixed; on the other hand, in our approach, we rely on external specifications based on the choreography (MPST) and the FSMs usage is internalised (i.e. FSMs are automatically generated from a global type), therefore it does not affect program structures.

Several works study an extension of actors in multicore control flows. Multithreaded active objects [11] allow several threads to co-exist inside an active object and provide an annotation system to control their concurrent executions. Parallel Actor Monitors (PAM) [23] is another framework for intra actor parallelism. PAMs are monitors attached to each actor that schedule the execution of the messages based on synchronisation constraints. Our framework also enables multiple control flows inside an actor as [11,23]. In addition, we embed monitors inside actors in a similar way as [23,11] embed schedulers. The main focus of the monitors in [11,23] is scheduling the order of the method executions in order to optimise the actor performance on multi-core machines, while our approach aims to provide explicit protocol specifications and verification among interactions between distributed actors in multi-node environments.

The work [22] proposes a solution to have multiple cooperatively scheduled tasks within a single active object similar to our notion of cooperative roles within an actor. The approach is implemented as a Java extension with an actor-like concurrency where communication is based on asynchronous method calls with objects as targets.

They resort to RMI for writing distributed implementations and do not allow specifying sequencing of messages (protocols) like ours.

The work [21] proposes a framework of three layers for actor roles and coordinators, which resembles roles and protocol mailboxes in our setting. Their specifications focus on QoS requirements, while our aim is to describe and ensure correct patterns of interactions (message passing).

Comparing with the above works, our aim is to provide effective framework for multi-node environments where actors can be distributed transparently to different machines and/or cores.

7 Conclusion

We propose an actor specification and verification framework based on multiparty session types, providing a Python library with actor communication primitives. Cooperative multitasking within sessions allows for combining active and reactive behaviours in a simple and type-safe way. The framework is naturally integrated with Celery [8] which uses advanced message queue protocol (AMQP [5]), and uses effectively its types for realising the key mechanisms such as the actor discovery. We demonstrate the overhead of our implementation is very small. We then show that programming in session actors is straightforward by implementing and rewriting usecases from [4]. To our best knowledge, no other work is linking FSMs, actors and choreographies in a single framework. As a future work, we plan to extend to other main stream actor-based languages such as Scala and Erlang to test the generality of our framework. As actor languages and frameworks are getting more and more attractions, we believe that our work would offer an important step for writing correct large-scale actor-based communication programs.

Acknowledgement We thank Ask Solem for his support and guidance. This work has been partially sponsored by VMWare, Pivotal and EPSRC EP/K011715/1 and EP/K034413/1.

References

1. Cell - actors for celery. `http://cell.readthedocs.org/`.
2. G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.
3. G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7:1–72, 1 1997.
4. Akka - scala actor library. http://akka.io/.
5. Advanced Message Queuing Protocol homepage. `http://www.amqp.org/`.
6. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
7. L. Caires. Spatial-behavioral types for concurrency and resource control in distributed systems. *Theor. Comput. Sci.*, 402(2-3):120–141, 2008.
8. Celery. `http://http://www.celeryproject.org//`.
9. S. Crafa. Behavioural types for actor systems. `arXiv:1206.1687`.

10. S. J. Gay, V. T. Vasconcelos, A. Ravara, N. Gesbert, and A. Z. Caldeira. Modular session types for distributed object-oriented programming. In *POPL*, pages 299–312. ACM, 2010.
11. L. Henrio, F. Huet, and Z. István. Multi-threaded active objects. In *COORDINATION*, volume 7890 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2013.
12. K. Honda, R. Hu, R. Neykova, T.-C. Chen, R. Demangeon, P.-M. Deniélou, and N. Yoshida. Structuring Communication with Session Types. In *COB'12*, LNCS, 2012. To appear.
13. K. Honda, A. Mukhamedov, G. Brown, T.-C. Chen, and N. Yoshida. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.
14. K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, pages 273–284. ACM, 2008.
15. R. Hu, R. Neykova, N. Yoshida, and R. Demangeon. Practical Interruptible Conversations: Distributed Dynamic Verification with Session Types and Python. In *RV'13*, volume 8174 of *LNCS*, pages 130–148. Springer, 2013.
16. S. M. Imam and V. Sarkar. Integrating task parallelism with actors. *SIGPLAN Not.*, 47(10):753–772, Oct. 2012.
17. D. Mostrous and V. T. Vasconcelos. Session Typing for a Featherweight Erlang. In *COORDINATION*, volume 6721 of *LNCS*, pages 95–109. Springer, 2011.
18. R. Neykova, N. Yoshida, and R. Hu. SPY: Local Verification of Global Protocols. In *RV'13*, volume 8174 of *LNCS*, pages 358–363. Springer, 2013.
19. N. Ng, N. Yoshida, and K. Honda. Multiparty session C: Safe parallel programming with message optimisation. In *TOOLS'12*, LNCS, pages 202–218. Springer, 2012.
20. Ocean Observatories Initiative. `http://www.oceanobservatories.org/`.
21. S. Ren, Y. Yu, N. Chen, K. Marth, P.-E. Poirot, and L. Shen. Actors, roles and coordinators - a coordination model for open distributed and embedded systems. In *COORDINATION*, volume 4038 of *LNCS*, pages 247–265. Springer, 2006.
22. J. Schäfer and A. Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In *ECOOP*, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.
23. C. Scholliers, É. Tanter, and W. D. Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, 2014.
24. Scribble project home page. `http://www.scribble.org`.
25. Online appendix for this paper. `http://www.doc.ic.ac.uk/~rn710/sactor`.
26. S. Tasharofi, P. Dinges, and R. E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *ECOOP*, volume 7920, pages 302–326. Springer, 2013.
27. BETTY WG3 - Languages Survey. `http://www.doc.ic.ac.uk/~yoshida/WG3/BETTY_WG3_state_of_art.pdf`.