HAL
open science

Session-Based Role Programming for the Design of Advanced Telephony Applications

Gilles Vanwormhoudt, Areski Flissi

To cite this version:

Gilles Vanwormhoudt, Areski Flissi. Session-Based Role Programming for the Design of Advanced Telephony Applications. 11th Distributed Applications and Interoperable Systems (DAIS), Jun 2011, Reykjavik, Iceland. pp.77-91, 10.1007/978-3-642-21387-8_7 . hal-00609512

HAL Id: hal-00609512

https://hal.science/hal-00609512v1

Submitted on 13 Feb 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Session-Based Role Programming for the Design of Advanced Telephony Applications

Gilles Vanwormhoudt[1,2] and Areski Flissi[2]

[1] Institut TELECOM
[2] LIFL/CNRS - University of Lille 1 (UMR 8022)
59655 Villeneuve d'Ascq cedex - France
{Gilles.Vanwormhoudt,Areski.Flissi}@lifl.fr

Abstract. Stimulated by new protocols like SIP, telephony applications are rapidly evolving to offer and combine a variety of communications forms including presence status, instant messaging and videoconferencing. This situation changes and complicates significantly the programming of telephony applications that consist now of distributed entities involved into multiple heterogeneous, stateful and long-running interactions. This paper proposes an approach to support the development of SIP-based telephony applications based on general programming language. Our approach combines the concepts of Actor, Session and Role. Role is the part an actor takes in a session and we consider a session as a collaboration between roles. By using these concepts, we are able to break the complexity of SIP entities programming and provide flexibility for defining new ones. Our approach is implemented as a coding framework above JAIN-SIP.

1 Introduction

In recent years, telephony services have endorsed significant changes by integrating a variety of communication forms including video, text and presence while managing aspects like mobility, security, etc. This evolution has created a need for telephony applications to involve an increasing widely range of distributed entities with capacities for participating into multiple, heterogeneous, stateful and long-running interactions. This requirement, compounded with the intricacies of underlying communications, make the programming of new telephony applications a daunting task.

By supporting a rich range of communication forms, the 'Session Initiation Protocol' (SIP) has contributed a lot to this evolution and many advanced telephony applications are now SIP-based. For programming the entities involved in these applications, two main categories of approaches have been proposed over the years. In the first category, we find domain-specific languages (DSL) like LESS[12], SPL[8], ECharts[11], StratoSIP[9] to program specific kind of SIP entities such as routing server, end user-agent or back-to-back user agents. All these DSLs provide high-level concepts to hide the intricacies of the underlying SIP technologies but they are usually limited to coarse-grained and dedicated operations and therefore prevent from implementing arbitrary telephony services. The second category of approaches is based on general purpose programming

P. Felber and R. Rouvoy (Eds.): DAIS 2011, LNCS 6723, pp. 77–91, 2011.
© IFIP International Federation for Information Processing 2011

language and the providing of large, powerful and generic APIs or frameworks such as JAIN-SIP, SIP-Servlet and JAIN-SLEE for the Java language[1]. However, although they enable the programming of unrestricted SIP applications, these approaches provide little support to layer the design of entities that are involved into multiples sessions or participate into sessions with complex message flow, two requirements often meet in advanced telephony applications.

In this paper, our goal is to facilitate the development of SIP-based applications programmed with general-purpose languages. To do so, we provide the developer with a programming model that raises the abstraction level with Actor, Session and Role as key concepts. Our notion of role encapsulates one fragment of the behavior played by an entity similarly to other existing role-based programming approaches[10,4] but in our model this notion is specifically related with the notion of session of interactions to provide session-based role programming. The roles played by a SIP entity, which is represented by an Actor, depend automatically on the sessions it participates in at runtime. Advantages provided by this model is to simplify the reasoning on entities description, to achieve a better modularization between session-dependent parts of entities and to improve the capacities for constructing SIP entities from reusable components. In addition to this model, we propose a lightweight implementation over JAIN-SIP that supports the definition of the proposed notions through a set of Java annotations.

The rest of this paper is organized as follows. After pointing out some issues underlying the design of advanced SIP applications in the next section, section 3 presents our programming model. Section 4 illustrates the model and discusses its benefits. Section 5 describes a lightweight implementation of the model in Java. Section 6 presents related works prior to conclude with section 7.

2 Issues in SIP-Based Applications Design

For supporting applications ranging from simple VoIP routing or instant messaging to sophisticated multimedia sessions involving multiple parties with presence management, SIP provides a rich range of communications forms. Communications can be stateless for simple messages exchange, session-based to exchange messages over a period of time or event-based to propagate information like state-change of an entity. One main benefit of SIP is that it enables mixing its communication forms to design advanced telephony applications. As an example of such application, which will serve in the following, we propose to consider an application that manages presence-based redirection[2] of invitation to a dialog combining voice and text. In this application, a server manages the incoming invitation for registered users depending on their status: when available, the server replies with the current user's address to directly invite him, otherwise the invitation is rejected. Figure 1 shows the architecture of this application and a use case of presence-based redirection leading to a successful dialog. In this figure, Alice updates its status because she leaves its office for a meeting (1,2).

[1] http://java.sun.com/products/jain/

[2] In SIP, redirection consists in directing the client to contact an alternate address.

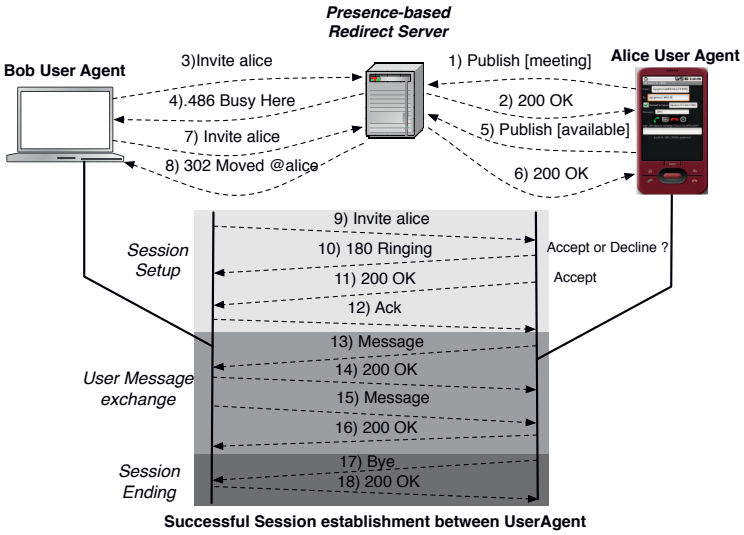

Fig. 1. Example of Presence-based Redirection

Then, Bob calls Alice during her meeting and he receives a BUSY response (3,4). When Alice becomes available (5,6), a new invitation from Bob results in a redirect response including Alice's address to initiate a dialog (7,8). Thanks to the returned address, Bob can directly contact Alice to establish a successful dialog (9 to 18).

Despite SIP features for supporting multiple communications forms, developing advanced applications like the previous one remains a complex task. In the following, we give three of the typical issues that complicate the programming of SIP entities.

1) Complex messages flow within a session : Within a session, SIP entities exchange and handle messages at each end. The handling of these messages consists generally in checking that the received message is valid with the entity's current state, performing actions and then changing to next state. There are several difficulties related to the handling of messages in a SIP application.

– A first difficulty is that messages can be received at any time. This entails that SIP entities may be prepared to react to any received messages, including while others are processed. In general, this is achieved by adopting an event-driven approach for message handling.

– Another difficulty related to message handling is that interpretation of messages in the flow is generally state-dependent. In the example, OK responses have a meaning that depends upon the current state or the previous request. Distinguishing between these interpretations generally require that the application maintains a session state according to the exchanged messages.

– Sometimes, the messages flow of a session may contain messages that are related to distinct concerns. For each concern, the related messages are not necessarily in separate sequences but may be interleaved with those from others concerns. In the example, this is illustrated in the lower part of Figure 1 where

we can identify three groups of messages corresponding to session setup, user message exchange and session ending concerns.

– A last difficulty is that message flow can be inverted during the session. Illustrations in the example are given by MESSAGE and BYE requests. This implies that entities must be able to provide behaviours for handling session flow in both directions and to act both as requestor and responder.

Because of the difficulties described previously, the behaviour of entities handling some part of the session flow like the previous one is usually not easy to design and there is a need to provide the developer with appropriate abstractions and decomposition mechanisms to facilitate the handling of messages related to a particular concern or a particular state inside the code.

2) Multi-branches session : A SIP session is not always restricted to a peer-to-peer conversation. There are some classical calling or presence scenarios which involve more than two peers in the same session. In these scenarios, one or several SIP entities generally act as facilitator between several participants to the session. For such an entity, this entails to handle more that one conversation within the same session and to coordinate all the conversations in order to serve the communication partners properly. In our example, we initially assume a redirect behaviour for the server but we can easily imagine changing this behaviour to a proxy-like one in order to extend the status of a user according to its participation in an existing dialog[3]. Compared to the previous behaviour, adopting this change requires that the server manages and coordinates two communication paths within a session : one to the callee and one to the caller, switching back and forth being a client and a server at the same time. When a SIP entity is involved in more than one conversation within a session, the complexity for describing its behaviour is inherently increased. To help the developer designing behaviour for multiple interwoven conversations, mechanisms should be provided. Such mechanism should enable expressing the state and behaviour related to each conversation separately but also simplify their coordination.

3) Multi-sessions management : The need to handle multiple sessions is another situation than makes the development of SIP applications and their entities complicated. Two cases may be distinguished for multi-sessions management. The first case is the one where the sessions managed by a SIP entity have the same type and typically occur concurrently. We generally encounter this case when designing SIP server. In our example, the redirect server is an illustration of this case as it must be able to manage calling session coming from multiple requestors. Here, the difficulty is that multiple session states must be maintained separately and concurrently. The second case is the one where each session has a different type. This case can be found particularly in the development of rich user agents and rich servers. In our example, Alice's user agent illustrates this case as it must be able to manage several sessions related to registration, incoming invitation and status modification. For this case, an additional difficulty besides maintaining multiple states is that each session may require the handling

[3] Proxy behaviour in SIP consists in relaying each request and response between user agents of a session.

of distinct set of messages resulting in a significant amount of messages to handle. Note that a combination of the two cases may sometimes be required for the design of a SIP entity as it is illustrated by the redirect server. For SIP entities, supporting multiple sessions makes their design more complicated because they must include characteristics for many states and behaviours. Therefore, some facilities are needed to limit the complexity. Such facilities should make possible to describe and manage part of the entity related to each session separately from others while simplifying the selection of the appropriate parts during interaction.

Regarding the previous issues, it appears that existing SIP APIs or frameworks do not provide abstractions to solve them. Because of this lack of abstractions, it is common for application developers using SIP frameworks to express the multiple states and behaviours related to parts like concerns, conversations or sessions in a monolithic way and manage them manually with intricate and scattered if-statements. This greatly complicates the application development task and results in cluttered code that is harder to understand, reuse and maintain. To cope with these issues and fill the gap, we present our role-based programming model in the next section.

3 Programming Model with Actor, Session and Role

As explained previously, development of advanced SIP applications is complex, leading to intricate structure of behaviour for SIP entities. In this section, we propose a programming model based on Actor, Session and Role concepts to help this development. By relying on these concepts, we are able to break the complexity of actor behaviour and to define this behaviour in a flexible way.

In our model, an actor is a top level component which represents a distributed SIP entity. During its lifetime, an actor is involved into one or several SIP sessions that may have the same nature or being different and exist concurrently. To handle sessions programmatically, we define two related concepts : session and session part. The session concept is orthogonal to actors : it aims to group the definition of roles that characterize a specific session. The session part concept establishes the link between an actor and a specific session : it aims to define one or several roles played by an actor when it participates to a particular session. Role concept is the basic unit of our programming model. It is used to describe one of the behaviour involved in a session. Within a particular session, an actor communicates with other actors that play dual or consistent roles.

Figure 2 summarizes the idea of actor, session and role concepts and shows capabilities of the model. Related actors A1 and A2 participate to a SIP session S1. Within this session, actor A1 plays a single role R1 and communicates with A2 that plays dual roles R2 and R2'. Figure 2 also shows a second SIP session S2 independent from S1 that involves actors A3 and A4 but also A1, already engaged in S1. For this session, we may observe that actor A1 plays a role R1' distinct from R1 played in S1. Actor A3 plays two distinct roles to communicate with A1 and A4 actors.

The combined use of actor, session and role proposed by our model allows to deal with the issues identified above. The first issue can be managed by

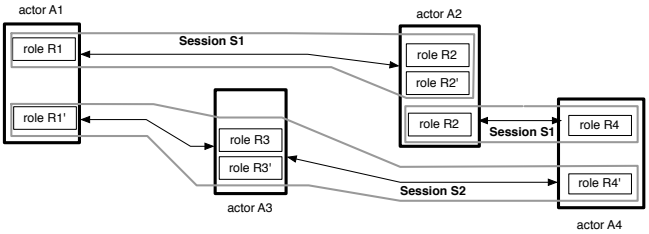

Fig. 2. Illustration of Actors, Sessions and Roles

defining multiple roles for a particular session (e.g. A2), each role dealing with one session concern. The capacity for an actor to play multiple roles for distinct communication path (e.g. A3) of a session may be exploited to solve the second issue. Concerning the last issue, the solution is provided by the ability of an actor to participate in multiple sessions with separate roles (e.g. A2 and A4).

In the following subsections, we present the structure of each concept and how they relate. We also explain how session parts and their related roles are created and activated during the actor life-cycle.

3.1 Session and Role

In our programming model, sessions are defined independently of actors. A session definition is a kind of module construct containing declarations of interacting roles types that are intended to be played by participating actors. There are two ways to achieve the declaration of roles types inside the session definition : by nesting a role type definition or by importing a role type. This second way enables reusing role types existing in libraries. Role types of a particular session are declared independently of other sessions. Indeed, it is possible for two sessions to rely on a same imported role type if they need the same slice of behaviour.

Code 1 gives, with a Java-like syntax, the definition of CallingSession, a session from our example that defines role types for dialog establishment and user message exchange. We can see that this session definition declares three roles types, the first two (*Callee* and *Caller*) by nesting and the third (*MessageHandler*) by importing thanks to the 'includerole' keyword.

A role type describes one behaviour involved in a session. Such a behaviour consists in handling incoming and outgoing SIP messages from and to the other roles while realizing the relevant logic. We provide three fixed kinds of role type to specify their capacities in terms of received and sent messages and help the analysis and the correlation of role types :

– **Client role** represents an asymmetric role type that sends requests and receives related responses.

– **Server role** represents an asymmetric role that receives request and sends related responses.

– **Client_Server role** represents a symmetric role that sends and receives both requests and responses.

```
// CallingSession session
CallingSession {
 clientserver role Callee {
  onInvite(Request r) { ...}
  onBye(Request r) {...}
  sendBye() { ...}
  ...
 }

 clientserver role Caller {
  sendBye() { ...}
  sendInvite() { ...}
  onOk(Response r) {...}
  onRinging(Response r) {...}
  onBye(Request r) {...}
  ...
 }

 includerole library.MessageHandler
}
```

```
// RegisterSession session
RegisterSession {
 server role RegAcceptor {
  Timer timer;
  String contactName;
  InetAddress currentAddress;
  onRegister(Request r) { ...}
  ...
 }

 client role RegRequestor {
  Timer timer;
  InetAddress currentAddress;
  sendRegister() {...}
  onOk(Response r) {...}
  ...
 }
}
```

Code 1. Two examples of session definition

The definition of a role type is quite similar to a class in the sense that it may include attributes, operations and may inherit from another role type. Beside operations, a role type may also contain message handlers that are special operations designed to handle an incoming SIP request or response. These operations have a specific signature that matches the type of the SIP message.

In the example above, the *RegRequestor* role type of *RegisterSession* session is a 'client' role that registers current user's address sending the REGISTER request and handles its related OK responses. It owns a *sendRegister* operation and a *onOK* response handler for that purpose. The *RegAcceptor* role type is a dual 'server' role that has only a message handler to respond to REGISTER request. From sessions, it is possible to define actors that interact by playing the related roles.

3.2 Actor and Session Part

Actors are described using actor type. An actor type usually includes one or several session parts. Like a class, an actor type may also contain declaration of attributes and methods that can serve to share some common data or operations between session parts. Furthermore, to specify which session parts are created on the basis of a particular request, an actor type may also have a special block containing rules which specify a condition coupled with a reference to a session part. The meaning of a rule is that an instance of the corresponding session part will be created if the condition is verified for the current state of the actor and the received request.

Code 2 illustrates the actor type for presence-enabled user agents (Alice's one) discussed in Section 2. This actor type includes attributes for managing the history of callers. It also contains declaration of session parts for the three sessions this actor type takes part: *RegisterSession*, *PublishSession*, and *CallingSession*. Preceding the session parts, we find a 'sessionControl' block containing a rule

```
/* PresenceEnabledUserAgent actor type */
actor PresenceEnabledUserAgent{
  String contactName;
  List<SipAddress> callerHistory;
  void clearHistory()  ...

 sessionControl {
  when (isINVITE(req) &&
         !hasSession(CallingSession))
    activate SPM
   default activate Error
 }

 sessionPart SPR:RegisterSession {
  play (RegRequestor)
 }

 sessionPart SPP:PublishSession {
   play (PresencePublisher)
  }
```

```
/* Continued here */
sessionPart SPI:CallingSession {
 String callerName;
 String getCallerName() { ... }

 play (Callee, MessageHandler)
 extension Callee {
   onInvite (Request req) {
    callHistory.add(req.getFromAddr());
    callerName = req.getCallerName();
    getRole(PublishSession,PresencePublisher).
       setNewState(State.available);
    super(req);
   }
   onBye (Request req) { ... }
  }

  roleControl {
    when (isINVITE(req)) activate Callee
    when (isMESSAGE(req)) activate MessageHandler
  }
}  ... } /* End of actor type */
```

Code 2. Example of an actor type

to specify that the *SPI:CallingSession* session part should be created when an INVITE request is received. Other session parts are intended to be created explicitly as they only contain client roles.

A session part defines the participation of an actor type to a specific session in terms of played roles. In a session part, this participation is specified by referencing the targeted session and declaring which roles defined in the session is played by actors of this type. If we take a look at the actor type given above, we can see that the session part named *SPI* is connected to the *CallingSession* defined previously. For this session part, two roles are specified using the 'play' keyword : *Callee* and *MessageHandler*. As illustrated by this example, introduction of attributes (*caller*) and methods to share common data and operations between played roles is also possible.

Similarly to actor type, each session part may include a special block containing rules for determining the roles to create on a particular request and the current actor state. Rule conditions can use predefined boolean functions to query the roles of the session. In Code 2, we have an example of this block introduced by the 'roleControl' keyword. This block contains two rules to state that the *Callee* (resp. *MessageHandler*) role should be played on reception of an INVITE (resp. MESSAGE) request.

In addition to the previous elements, a session part may also introduce extension of roles defined for the referenced session and played by the actors. A role extension enables refining its behaviour. Such extension may be needed to add interactions between sessions inside a role or between roles of a session. This capacity is used in the *SPI* session part to extend the behaviour defined in *CallingSession* for the *Callee* role. Here, this extension introduced by the 'extension' keyword is required to update the status of the user when he enters or leaves a

dialog. It is achieved by redefining the handlers attached to INVITE and BYE messages with operations to interact with *PresencePublisher* role.

3.3 From Concept to Runtime Entities

In our programming model, actor type, session parts and role types are instantiated to form the state and behaviour of an actor. At runtime, a session part must be considered as the reification of a session from an actor's point of view.

Actors are created from actor type and represents SIP entity at runtime. During the lifetime of an actor, session parts are instantiated to reflect its participation in real SIP sessions. This instantiation can be done explicitly on demand by means of a new-like construct or occurs implicitly on the basis of received request. For a receiving request, the decision to create a session part instance depends if there already exists a session part matching the session-id included in the request. If none exists, rules attached to the actor for session parts are evaluated to create a matching session. A session part may be instantiated more than once per actor. This corresponds to the situation when an actor participates to several sessions of the same kind during its lifetime.

At runtime, each session part instance representing a real session aggregates roles that are played by the actor. The creation of roles attached to a session can be done in two ways, like session part instantiation: either explicitly by means of a new-like construct or implicitly on the basis of a received request. In the latter case, rules attached to the session part are evaluated to determine if a corresponding role must be created.

Message handlers provided by roles are automatically executed on the basis of incoming messages through a forwarding process from actor. When an actor is requested to handle an incoming SIP message, the forwarding process is based first on a selection of the session part instance matching the session-id of the request[4]. After that, the processing continues by choosing a role attached to the selected session or by creating a new one if necessary. At last, the message is forwarded to the role by triggering its corresponding message handler.

4 Revisiting the Example

To illustrate our approach, we propose to revisit our example in terms of Actors, Sessions, Session parts and Roles. Figure 3 shows the resulting architecture which is composed of three actor types: *PresenceEnabledUA*, *RedirectServer* and *UserAgent*. These actors are involved in four sessions which are represented by horizontal boxes crossing actors. The first session, named *RegisterSession*, takes place when a *PresenceEnableUA* actor registers its current address to the server. *PublishSession* is the session to update the user status. The *LocateSession* is initiated when an external UserAgent actor communicates with the *RedirectServer* to locate a registered user agent (*PresenceEnabledUserAgent*). Finally,

[4] As explained before, the incoming message may entail a creation of session instance.

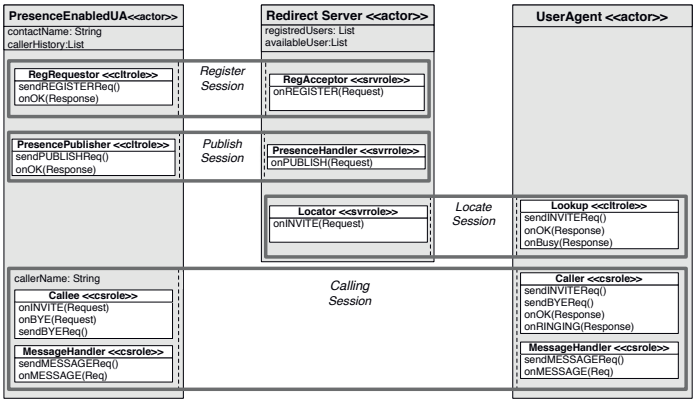

Fig. 3. Architecture of our example with Actors, Sessions and Roles Types

CallingSession is the session for handling the dialog between *PresenceEnabledUA* and *UserAgent* actors types. For the above sessions, we can see that the involved actors play separate roles which are figured by class-like box (session parts are not shown but just indicated by dotted lines).

Figure 4 shows the relationships between the flow of SIP messages and instances of session parts and roles for a dialog setup between Alice and Bob. After receiving the response containing Alice's address from the server, the Lookup role instance played by Bob's *UserAgent* actor explicitly creates a new CallingSession-related session part and an associated *Caller* role. Next, this role is invoked to send the INVITE message establishing the real session. When the Alice's *PresenceEnabledUA* actor receives the message, it detects that there is no matching session, so it evaluates its rules for determining the session part to create and activate. In the current case, it is a CallingSession-related session part which is instantiated. Then, this session part instance evaluates its own rules for determining which role must be played by the actor from the current message and this kind of session. The result is the creation of a *Callee* role which is finally invoked for processing the message. When playing the *Callee* role, the *PresenceEnabledUA* actor returns an OK response to the UserAgent actor for the same session. Because the *UserAgent* actor is already active in this session and already has a *Caller* role to handle OK response, no new role is created and this role is selected to process the responses and confirm the dialog establishment with an ACK request. After dialog establishment is complete, both actors can play the *MessageHandler* role that deals with sending and reception of user messages inside the session.

Through this example, we have illustrated how our model supports the decomposition of behaviour into multiple sessions parts with their respective roles and accomplishes some properties to cope with the issues and requirements identified in section 2. This model also offers the following advantages :

– It raises the level of abstraction as the developer can think about the behaviour of its actor at a higher level thanks to session part and role. This

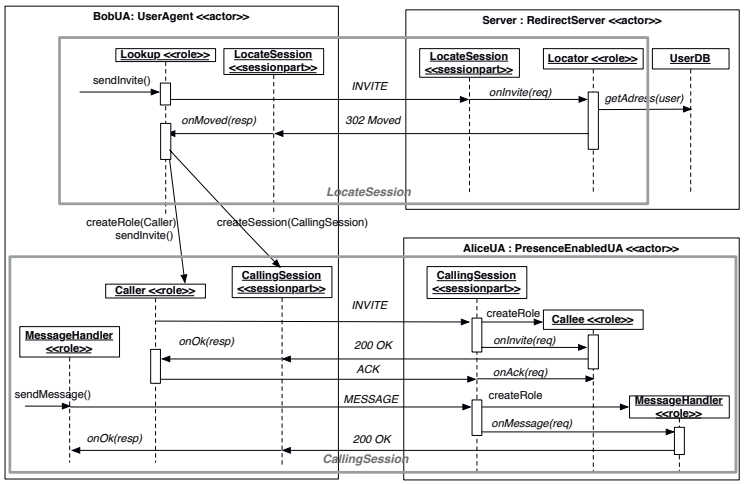

Fig. 4. Relationships between SIP messages and actor components

contrasts with conventional SIP frameworks which require to examine implementation code in detail to get a similar view.

– The encapsulation of behaviour and state into multiple session parts and role components with delimited scope contributes to increase the modularity of actors. As a result of this enhanced modularity, maintenance and evolution of SIP entities and services are made easier. This modularity allows, for instance, to change the server with a proxy behaviour by just replacing its *Locator* role with two coordinated forwarder roles supporting multi-branches communications.

– Automatic selection of sessions and roles as well as automatic messages forwarding to roles allow to reduce the coding effort since number of controls and extra-state to ensure execution of the appropriate behaviour is minimized compared to SIP frameworks.

– Finally, our approach gives the ability to capture some recurrent behaviour into reusable role types and reuse them by inclusion into sessions. This is an main advantage over existing DSLs for SIP where the question of reusability is generally eluded and over SIP frameworks where reusability is limited by the hardwiring of session-related parts into methods.

5 A Coding Framework above JAIN-SIP

We have implemented our approach in Java through a coding framework which is presented at Figure 5. Actor-annotated Java code is transformed to produce executable actors, that is to say Java classes that specialize an actor framework. The main classes of the actor framework are *SipActor*, *SipSessionPart*, *SipClientRole*, *SipServerRole* and *SipClientServerRole*. These classes implement the proposed concepts and a runtime engine above JAIN-SIP.

To leverage building of SIP applications using our framework, a set of annotations presented in Table 1 has been defined. A class with the `@actor` annotation

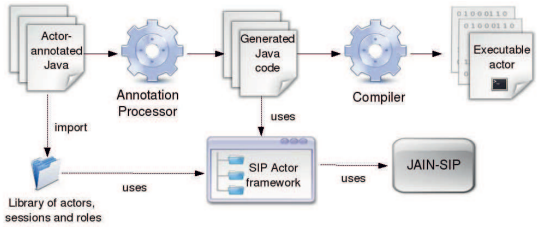

Fig. 5. Coding framework architecture

Table 1. Annotations for programming SIP Actors in Java

Annotation	Attributes	Description
`@actor`	-	Declare an Actor class
`@session`	-	Declare an abstract Session class
`@sessionPart`	type::Class	Declare a Session part class
`@clientserverRole`	-	Declare a symetric role class
`@clientRole`	-	Declare an asymetric role class
`@serverRole`	-	Declare an asymetric role class
`@useSessionPart`	type::Class, method::String, condition::String	Specify an activation clause for session part
`@useRole`	type::Class, method::String, condition::String	Specify an activation clause for role
`@includeRole`	type::Class	Import an existing role in a session class

will inherit from the *SipActor* class of the framework, whereas a class annotated with `@sessionPart` will inherit from *SipSessionPart* class. The `@session` annotation is used with an abstract class to define a session. Such a class can declare inner role classes to define the related role types or use the `@includeRole(type)` annotation to import existing role classes from a library. The three kinds of role types provided by the model are declined in corresponding annotations for classes: `@clientRole`, `@serverRole`, `@clientserverRole`.

An actor declares its sessions part with `@useSessionPart`(*type*, *method*, *condition*) annotation on its class. The mandatory *type* attribute determines the session part class that has to be imported and activated. The mandatory *method* attribute provides the type of SIP request method (*e.g.* `INVITE` message) that triggers the activation of the session part. The optional *condition* attribute is a string that refers to the name of a boolean method that describes some particular conditions for the activation of the session part. The principle is similar for declaring roles of a session part, thanks to `@useRole` annotations, except that it is used with a session part class. Code 3 gives the annotated version of *CallingSession* session and *PresenceEnableUserAgent* actor type.

A preliminary study on performance was conducted using the SIPp traffic generator and a JAIN-SIP and framework version of the same user agent server. Results were produced for 1000 calling sessions with a rate of 5/sec. These results show that the framework overhead is about 10 percent compared to JAIN-SIP which is relatively low given that the framework is not yet optimized.

```
@session  // Declaration of session
public abstract class CallingSession {
  @clientserverRole  // Declaration of role class
  public class Callee {
    public void onInvite (Request req, ServerTransaction tx) { ... }
    public void sendBye () {...}
  } ...
}
```

```
@sessionPart // Declaration of session part
@useRole(type=ExtCallee.class,method="INVITE")
@useRole(type=MessageHandler.class,method="MESSAGE")
public class CallingSessionPartPUA {
  @extension(Callee.class) // Extension of role class
  public class ExtCallee {
    public void onInvite (Request req, ServerTransaction tx) { ... } ...
   }
}
```

```
@actor // Declaration of actor class
@useSessionPart(type=RegisteringSessionPartPUA.class)
@useSessionPart(type=PublishingSessionPartPUA.class)
@useSessionPart(type=CallingSessionPartPUA.class,method="INVITE",condition="hasNoCallingSsn"))
public class PresenceEnableUA {
   public boolean hasNoCallingSsn() { ... }
}
```

Code 3. Example of annotations use to code the presence server example

6 Related Works

In earlier role-based programming approaches, roles are meant to capture the dynamic and temporal aspects of real worlds objects and the view adopted for roles is object-centric: roles are defined as being independent from the interaction. From an object-centric view, our role model have similarities with some earlier approaches and their variants [10,4] : it enables multiple role instances for a player and it links the roles to its player using aggregation relationships. However, some features like the distinction between client and server roles, the event-based triggering of roles and the capacity to have roles running in parallel threads are also unique to our role model.

A few works combine roles with a notion of context to express the fact that the interaction possibilities change according to the properties of the interacting objects. In powerJava [1], roles represent the possibilities offered by an object to interact with it. For a client, the interaction with such an object is made by acquiring one of its offered role. Similar ideas have been proposed in ActorFrame [2], a Java-based framework for the design of distributed services with actors and roles. The work described in [6] presents a programming model with context-dependent roles for actors. Roles represents adaptations of an actor that are automatically selected for each message based on the context of the message sender and receiver. In our approach, contexts for roles are provided by sessions. This make our approach different by two main points: selection of a role is not controlled by an interacting entity but only by the owner (actor) and our notion of context is long-lived, i.e persistently activated between messages.

The notion of role is also related to the concept of collaboration that aims to describe the interactions among different objects in a given context. ObjectTeam/J [5] and EpsilonJ [7] are extensions of Java that group roles which interact into collaboration modules. Roles inside a collaboration module are played by objects of base classes either explicitly or implicitly, enabling them to interact. In our approach, sessions and session parts provide similar capacities with the main difference that they are supported in a distributed context.

Finally, authors of [3] have proposed the notion of session type as a language construct to support description and type-checking of protocols between parallels threads. A session type is implemented in dual operations of interacting components using a correlated sequence of receive and send instructions. Compared to the use of one or more role for describing the behaviour of a component related to a session, an operation implementing a session-type is more fine-grained and offers a concise and clearer view of the interaction structure but they also provide less capacities for reuse and flexible composition of behaviour.

7 Conclusion

To tackle some issues arising from the design of advanced telephony applications based on SIP, we have proposed an approach that enables involved entities to be constructed as actors playing roles in multiples sessions. Proposed approach and its implementation have been experimented through the development of various SIP entities such as user-agents, third-party agents as well as presence and proxy servers. Currently, we are working on elaborating a library of reusable roles from these experiments. In future works, we plan to enhance the programming model with inheritance for sessions and actor types and event-based mechanisms to support coordination between roles and between session parts. We also plan to explore the extensibility of SIP for typing exchanged messages with sessions and roles to enable dynamic roles alignment and synchronization between actors. A last perspective is to better integrate our concepts with the host programming language by using capabilities of some languages for embedding DSL.

References

[1] Baldoni, M., Boella, G., van der Torre, L.: Interaction among objects via roles: sessions and affordances in java. In: 4th Int. Symp. on Principles of Programming in Java (2006)

[2] Bræk, R., Melby, G.: Model-Driven Service Engineering. In: Model-Driven Software Development. Springer, Heidelberg (2005)

[3] Dezani-Ciancaglini, M., Mostrous, D., Yoshida, N., Gairing, M.: Session Types for Object-Oriented Languages. In: Hu, Q. (ed.) ECOOP 2006. LNCS, vol. 4067, pp. 328–352. Springer, Heidelberg (2006)

[4] Graversen, K.B.: The nature of roles. A taxonomic analysis of roles as a language constructs. In Phd Thesis, IT University of Copenhagen (2006)

[5] Herrmann, S.: A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java. In: Applied Ontology, vol. 2. IOS Press, Amsterdam (2007)

[6] Vallejos, J., Ebraert, P., Desmet, B., Van Cutsem, T., Mostinckx, S., Costanza, P.: The Context-Dependent Role Model. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, pp. 1–16. Springer, Heidelberg (2007)
[7] Monpratarnchai, S., Tetsuo, T.: The Implementation and Execution Framework of a Role Model Based Language, EpsilonJ. In: Proceedings of SNPD 2008 (2008)
[8] Palix, N., Consel, C., Reveillere, L., Lawall, J.: A stepwise approach to developing languages for SIP telephony service creation. In: Proceedings of IPTComm 2007 (2007)
[9] Zave, P., Cheung, E., Bond, G., Smith, T.: Abstractions for Programming SIP Back-to-Back User Agents. In: Proceedings of IPTComm 2009 (2009)
[10] Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowledge Engineering 35 (2000)
[11] Smith, T., Gregory, G., Bond, W.: ECharts for SIP Servlets: a state-machine programming environment for VoIP applications. In: Proceedings of IPTComm 2007 (2007)
[12] Wu, X., Schulzrinne, H.: Handling feature interactions in the Language for End System Services. In: Feature Interactions in Telecommunications and Software Systems VIII (2005)