Systèmes Multi-agents

Connexion Application - SMA

Claude Moulin

Université de Technologie de Compiègne

IA04

Sommaire

Sommaire

Objectifs

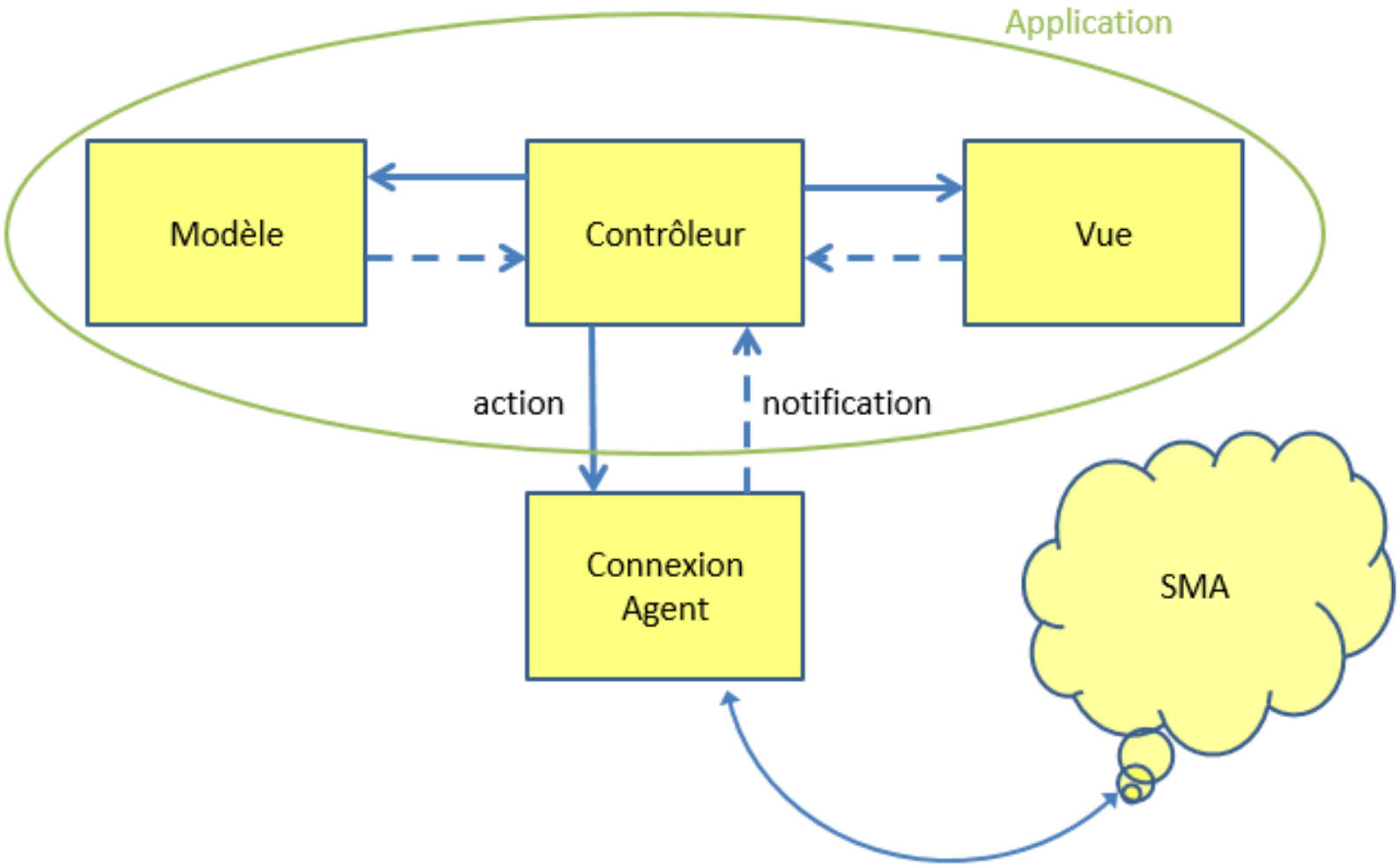
- Structuration des accès à un SMA, entrée et sortie.
- Généricité la plus élevée possible.
- Application SMA :
  - Technologie JADE.
  - On suppose qu'un SMA est actif dans un Intranet.

Sommaire

Principe

- On considère une application standalone qui doit accéder à un SMA et qui doit être notifiée lorsque le SMA produit des informations qui la concerne.
- L'application est bâtie sur une architecture Modèle Vue Contrôleur.
- Il faut créer un agent chargé d'être l'intermédiaire entre l'application et le SMA.
- L'application doit créer un container d'agents et l'agent connexion.

Modèle MVCA


Application
Modèle
Contrôleur
Vue
action
notification
Connexion
Agent
SMA

Structure MVC

- L'application crée la vue et le contrôleur.
- Le contrôleur crée le modèle et initialise les binding entre la vue et le modèle.
  - Installation de fonctions de réponse aux événements issus de la vue.
  - Installation de fonctions de réponse aux modifications des propriétés du modèle.
- Le contrôleur possède une référence sur la vue et sur le modèle.
  - Pour activer une méthode du modèle à partir d'une action sur la vue.
  - Pour modifier l'interface à partir d'une modification du modèle.

Structure MVCA

- Le contrôleur doit avoir une référence sur l'agent pour activer un envoi de message à partir d'une action sur l'interface et pour installer des fonctions de réponse aux réception de message par l'agent (binding agent).
- L'application crée le conteneur d'agents et l'agent connexion en passant le contrôleur en paramètre de contruction de l'agent (on ne peut avoir une référence sur l'agent que dans l'agent lui même).
- Le constructeur de l'agent référencie l'agent dans le contrôleur (setAgent).
- Le contrôleur initialise le binding avec l'agent.

Application

```
public void start(Stage stage) throws Exception {
 FXMLLoader loader = new FXMLLoader();
 loader.setLocation(getClass().
       getResource(<fichier fxml>));
 Pane root = (Pane) loader.load();
 ConnexionToJadeController controller =
       loader.getController();
 createAgentContainer(controller);
 stage.setTitle("Connexion - Jade");
 Scene scene = new Scene(root,500,300);
 stage.setScene(scene);
 stage.setOnCloseRequest(
                    evt -> stopAgentContainer());
 stage.show();
}
```

Application : Conteneur d'agents

```
public void createAgentContainer(
            ConnexionToJadeController controller) {
 Runtime rt = Runtime.instance();
 ProfileImpl p = null;
 try {
  p = new ProfileImpl(SECONDARY_PROPERTIES_FILE);
  ContainerController cc = rt.createAgentContainer(p);
  AgentController ac = cc.createNewAgent("jade",
    "jade.agent.ConnexionJadeAgent",
              new Object[]{controller});
  ac.start();
  containerController = cc;
 } catch (Exception ex) {
   ex.printStackTrace();
 }
}
```

Application : fermeture

On détruit le conteneur d'agent :

```
public void stopAgentContainer() {
 try {
   containerController.kill();
   Thread.sleep(500);
   System.exit(0);
 } catch (StaleProxyException e) {
      e.printStackTrace();
 } catch (InterruptedException e) {
      e.printStackTrace();
 }
}
```

Agent

L'agent récupère le contrôleur de l'application et s'y enregistre :

```
protected void setup() {
 super.setup();
 controller =
   (ConnexionToJadeController) getArguments()[0];
 controller.setAgent(this);
}
```

Binding agent

- L'agent possède des Property qui sont mises à jour par lui-même lorsqu'il reçoit des messages du système multi-agent.
- Le contrôleur de l'application doit mettre en place les fonctions de réponse à l'activation de ces Property.

Agent : property

```
StringProperty messageAnswer =
    = new SimpleStringProperty();
public class ExchangeSMABehaviour
                        extends AchieveREInitiator {
  ... // Constructeur avec message
@Override
   protected void handleInform(ACLMessage inform) {
    messageAnswer.setValue(setAnswer(inform));
   }
}
```

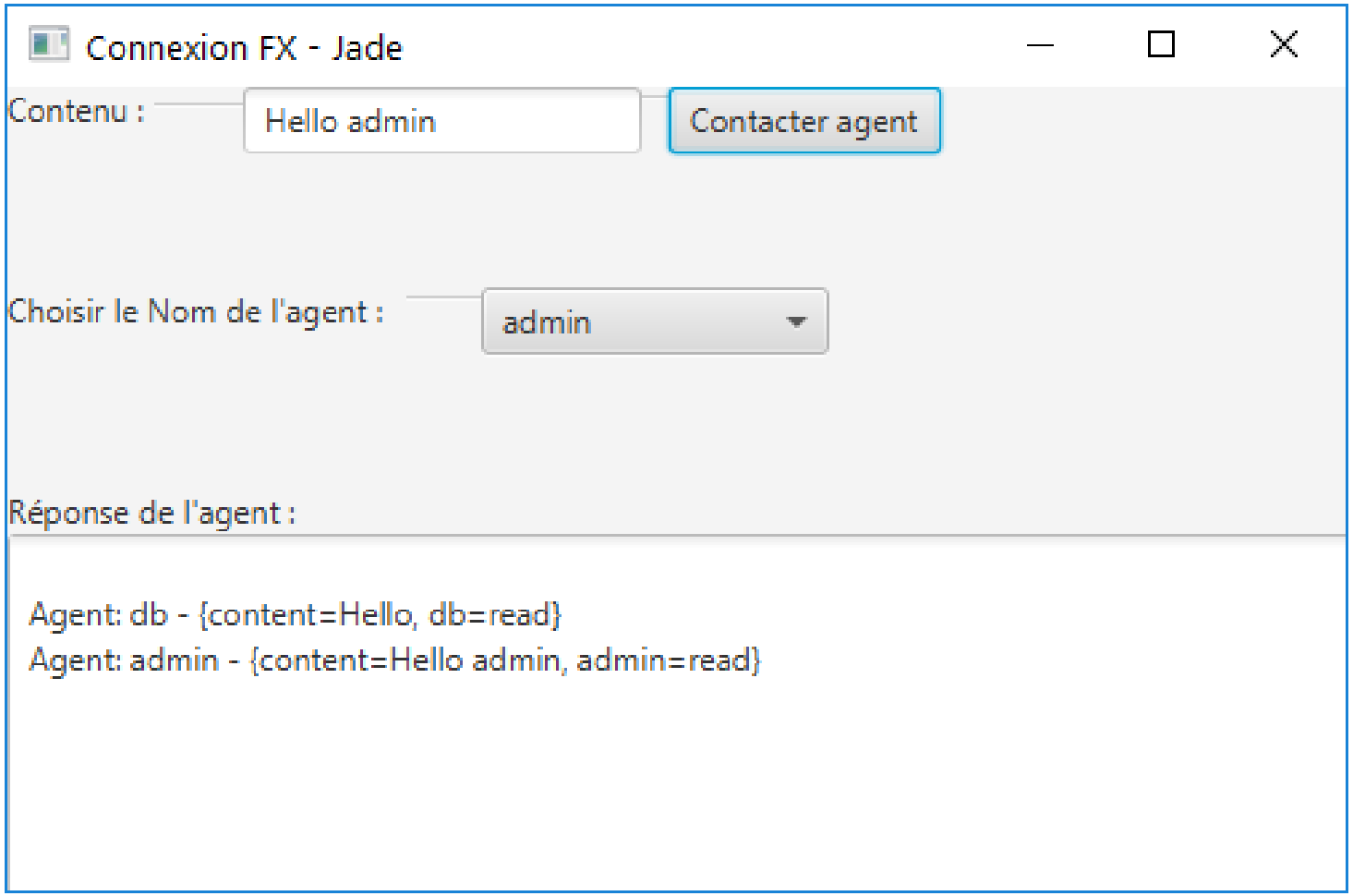
Contrôleur : Binding agent

```
public void setAgent(ConnectionJadeAgent agent) {
 this.agent = agent;
 initializeAgent();
}
public void initializeAgent() {
 agent.messageAnswerProperty().addListener(
       (obsv, oldv, newv) -> displayAnswer(newv));
}
private void displayAnswer(String message) {
 // met à jour l'interface
}
```

Utilisation de l'agent connexion

- L'agent connexion fait partie du système multi-agent.
- Il sait comment retrouver les agents du système et quels types de message envoyer.
- Le contrôleur de l'application doit activer une méthode de l'agent qui met en place un Behaviour pour communiquer avec le système.

Agent Connexion



Contrôleur : Appel agent

- Le contôleur de l'application active une méthode de l'agent

```
public void activateAgent() {
  activateAgent(content.getText());
}
private void activateAgent(String content) {
  if (receiverAgentName != null)
    agent.send2Agent(content, receiverAgentName);
}
```

Agent : installation behaviour

- L'agent installe un behaviour qui enverra un message à l'agent requis du système.

```
public void send2Agent(String content, String toAgent)
{
//Méthode pour créer le message
  ACLMessage message = formatRequest(toAgent,content);
  addBehaviour(new ExchangeSMABehaviour(this, message));
}
```

Sommaire

Principe

- Une application cliente a besoin d'utiliser un SMA tournant sur un serveur.
- Elle communique via un serveur REST, une SERVLET, une WebSocket, ou une base noSQL qui :
  - est installé(e) sur la même station que le SMA ;
  - met en place un agent Gateway chargé de communiquer avec le SMA.
- Elle contient un module assimilé à un agent.
- Elle n'a pas besoin d'utiliser la technologie Java.

Fonctionnement

L'application cliente envoie :

- des requêtes POST au serveur REST (bloquantes) ; Un contrôleur du serveur REST est chargée du traitement de la requête.
- des requêtes à une servlet chargée de les propager vers le SMA.
- des messages à l'endpoint d'une Websocket chargé de les propager vers le SMA.
- publie des messages sur un channel d'un serveur noSQL qui les propage aux subcribers.

Fonctionnement

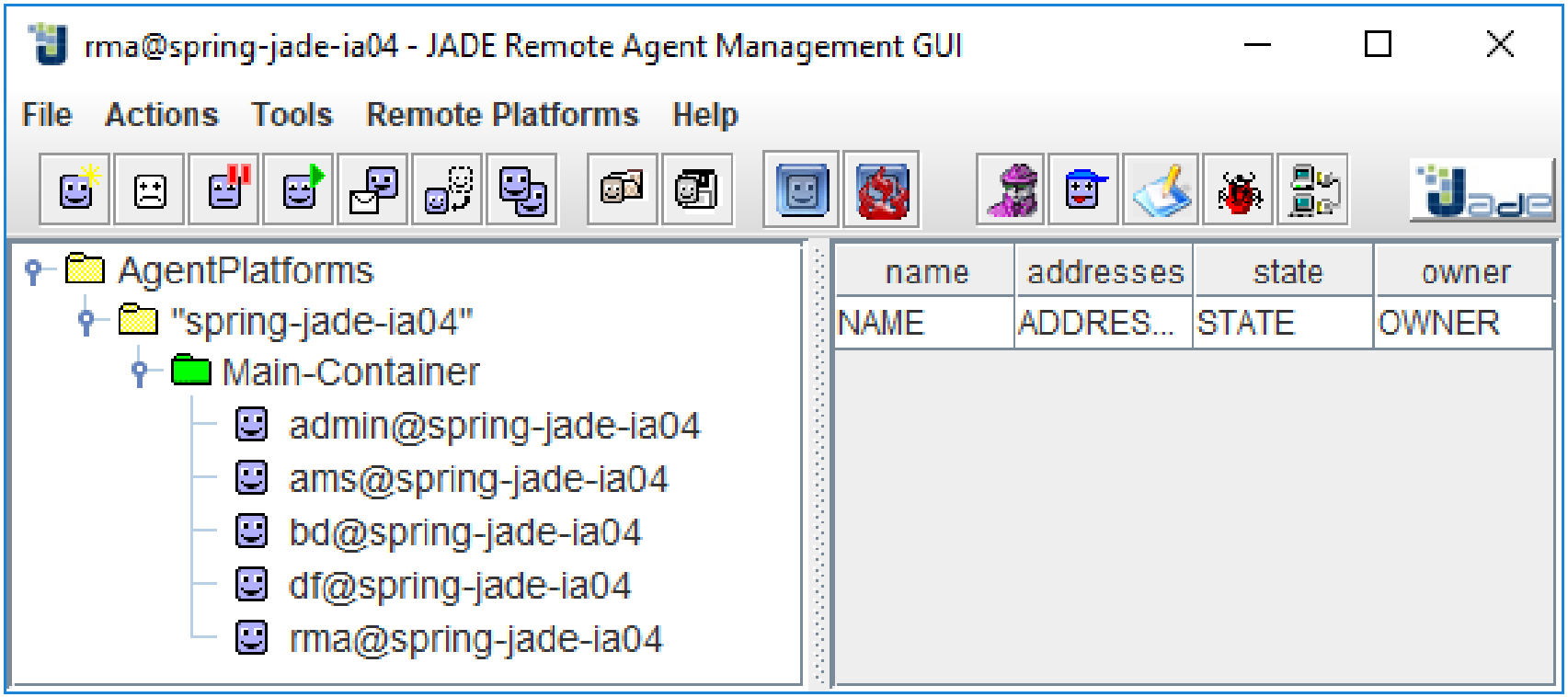
L'élément serveur (contrôleur REST, servlet, ...) chargé de la requête :

- active l'agent Gateway ;
- L'agent Gateway met en place un behaviour qui envoie un message à un agent ;
- Le behaviour attend une réponse au message puis s'arrête.

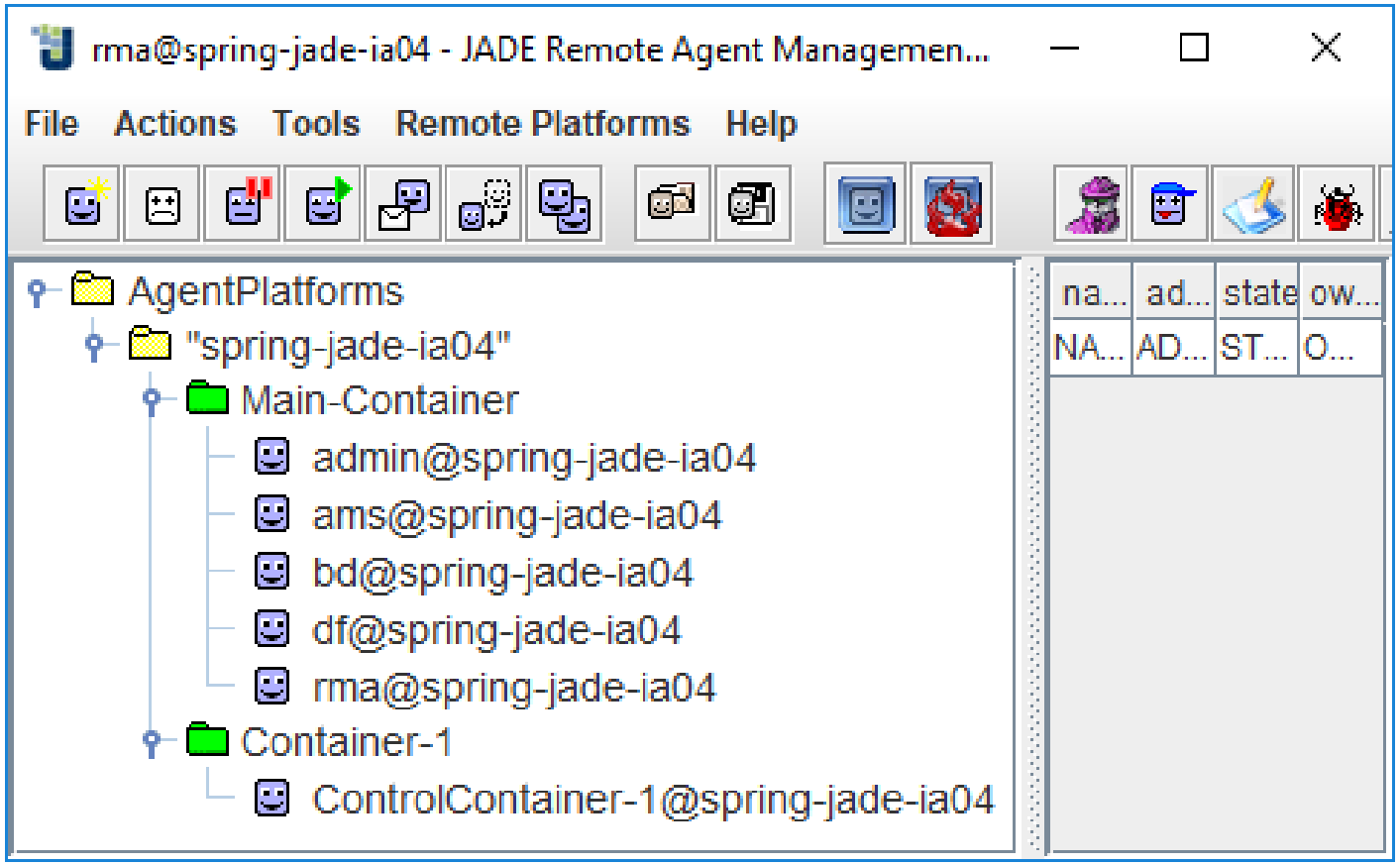
L'élément serveur :

- récupère les informations de réponse déposées dans le behaviour.
- structure la réponse.
- retourne la réponse à l'application cliente, ce qui la débloque le cas échéant.

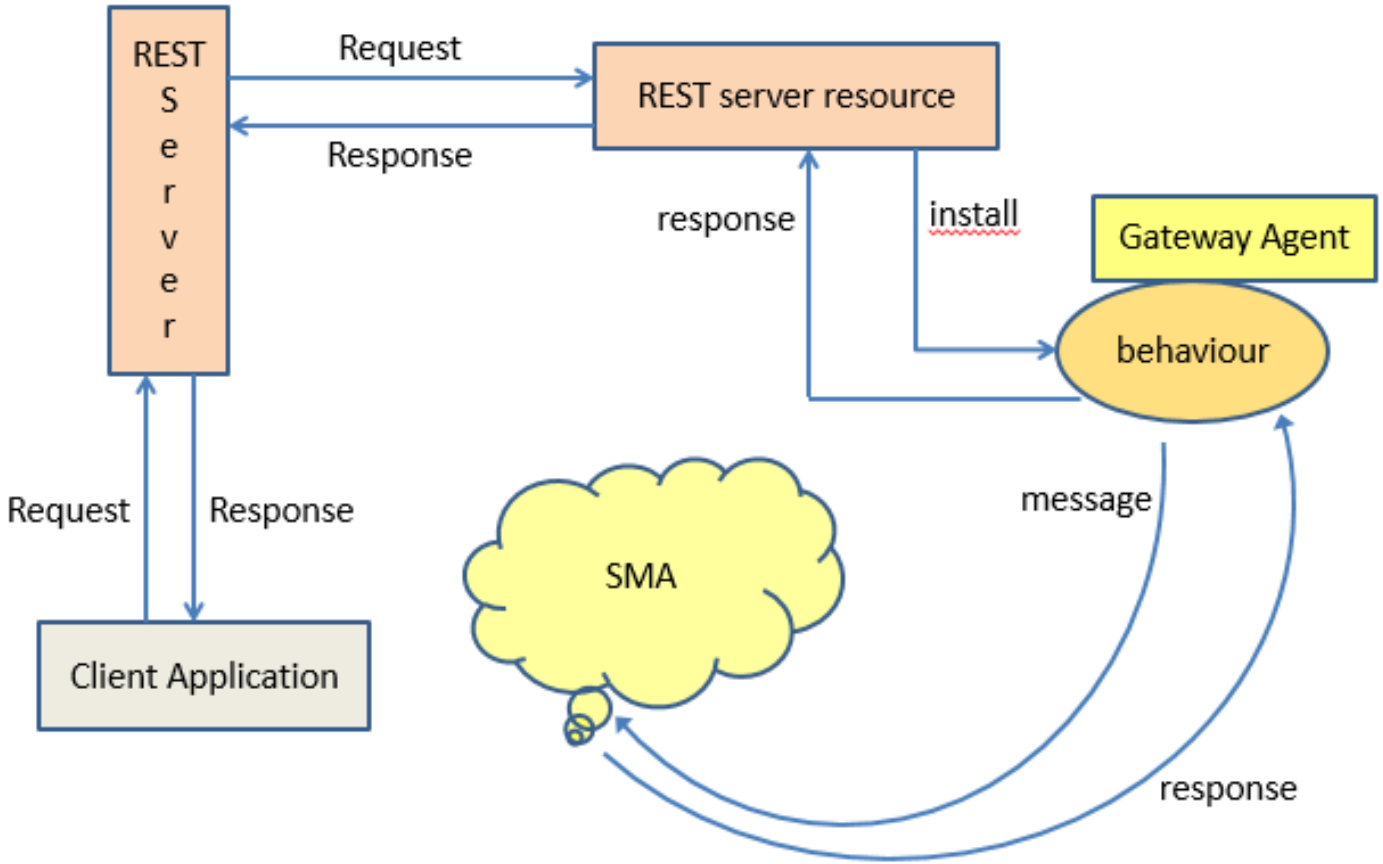
Agent Gateway



Agent Gateway



Agent Gateway



Destinataire de la requête

- L'agent simulé dans l'application cliente doit fournir les informations nécessaires au behaviour : données nécessaires à la construction d'un message.
- Agent destinataire :
  - L'agent simulé donne le nom de l'agent destination (ou un moyen de le retrouver).
  - Le serveur fournit différentes adresses pour différents types de requêtes vers le SMA.
  - Le serveur fournit une adresse paramétrée pour différents types de requêtes vers le SMA. En est déduit le nom de l'agent destination.
- Le protocole est un Request, mis en place par un AchieveREInitiateur.

Contenu et adressage

- On considère que le contenu de la requête POST est une chaîne JSON représentant un objet : `Map<String,String>`.
- `{"city":"paris","table":"users"}`
- On considère que l'adressage du serveur contient une entrée permettant de retrouver le nom de l'agent :
  - `"http://...:8182/db"` (agent db)
- On considère que l'adressage du serveur contient une entrée permettant de donner le nom de l'agent :
  - `"http://...:8182/agent/{aid}"`
  - `"ws://...:8025/websockets/sma/{aid}"`

Sommaire

Classes nécessaires

- Lancement de l'application contenant le serveur Rest
- Controleur Web qui gère l'adressage et les réponses
- Bean de customisation

Serveur Spring

Lancement du serveur :

```
@SpringBootApplication
public class JadeApplication {
 public static void main(String[] args) {
  SpringApplication.run(JadeApplication.class,
                        args);
 }
}
```

Configuration du port

(classe dans le même package que la classe serveur).

```
@Component
public class CustomizationImageBean implements
            EmbeddedServletContainerCustomizer {
 @Override
 public void customize(
    ConfigurableEmbeddedServletContainer container) {
  container.setPort(8182);
 }
}
```

Contrôleur Spring

- On considère :
  - que la requête contient une String, sérialisant une map contenant les paramètres nécessaires (ou un objet servant de contenu au message à envoyer à un agent).
  - qu'une adresse se termine par le nom de l'agent receveur du message.

Contrôleur Spring

(classe dans le même package que la classe serveur).
Requête : `http://...:8182/db`

```
@RestController
public class SpringJadeController {
 @PostMapping(value = "/db")
 public String forAgentDB(@RequestBody String content) {
  ...  // le contenu de la requête est obtenu par
       // l'annotation @RequestBody
 }
 ...
 private void activeAgent(Behaviour behaviour) {
  ...
 }
}
```

Contrôleur Spring

- On considère :
  - que la requête contient une String, sérialisant une map contenant les paramètres nécessaires (un objet contenu du message à envoyer à un agent).
  - qu'une adresse se termine par un paramètre aid définissant le nom de l'agent receveur du message.

Contrôleur Spring

(classe dans le même package que la classe serveur).
Requête : `http://...:8182/agent/admin`

```
@RestController
public class SpringJadeController {
  ...
 @PostMapping(value = "/agent/{aid}")
  public String forAgentId(@RequestBody String content,
                           @PathVariable String aid) {
   ...
}
 private void activeAgent(Behaviour behaviour) {
  ...
 }
}
```

Contrôleur Spring

```
@RestController
public class SpringJadeController {
 @PostMapping(value = "/agent/{aid}")
  public String forAgentId(@RequestBody String content,
                           @PathVariable String aid) {
   ProcessBehaviour behaviour = new ProcessBehaviour(aid,
                                                     content);
   activeAgent(behaviour);
   return behaviour.answer;
 }
 private void activeAgent(Behaviour behaviour) {
  ...
 }
}
```

Contrôleur Spring

```
@RestController
public class SpringJadeController {
 ...
  private void activeAgent(Behaviour behaviour) {
   try {
     JadeGateway.execute(behaviour);
   } catch (Exception e) {
     e.printStackTrace();
   }
 }
}
```

Behaviour

Le behaviour connait le destinataire et le contenu. Il doit :

- 1. Créer le message.
- 2. Envoyer le message.
- 3. Attendre le message réponse.
- 4. Fournir le contenu du message réponse comme réponse à la requête.

Un premier behaviour générique s'occupe des tâches 1 et 4 et met en place un AchieveREInitiateur qui réalise les tâches 2 et 3.

Behaviour

```
public class ProcessBehaviour extends Behaviour {
 String content;
 String receiverName;
 public String answer = "";
 Boolean messageToSend = true; // le message doit être envoyé
 Boolean done = false; // le behaviour est terminé
 public ProcessBehaviour(String agentName, String content) {
 ...
 }
 @Override
 public void action() {
 ...
 }
 @Override
 public boolean done() { return done; }
}
```

Behaviour - Constructeur

```
public class ProcessBehaviour extends Behaviour {
 ...
 public ProcessBehaviour(String agentName, String content) {
  super();
  this.content = content;
  this.receiverName = agentName;
 }
...
}
```

Behaviour - Méthode action

```
public class ProcessBehaviour extends Behaviour {
 ...
 @Override
 public void action() {
  if (messageToSend) {
   ACLMessage message = new ACLMessage(ACLMessage.REQUEST);
   message.setContent(content);
   message.addReceiver(
                  new AID(receiverName, AID.ISLOCALNAME));
   myAgent.addBehaviour(
      new ProcessInternalBehaviour(myAgent, message));
   messageToSend = false; // le message est envoyé
  }
 }
...
}
```

Behaviour interne

```
public class ProcessInternalBehaviour extends
                                AchieveREInitiator {
 public ProcessInternalBehaviour(
                    Agent agent, ACLMessage message) {
  super(agent, message);
 }
 @Override
 protected void handleInform(ACLMessage inform) {
   answer = inform.getContent();
   done = true;
 }
 @Override
 protected void handleRefuse(ACLMessage refuse) {
   answer = refuse.getContent();
   done = true;
 }
}
```

Client

- L'application cliente contient un module assimilé à un agent pouvant envoyer des requêtes HTTP (POST). Ex : utilisation d'un HTTPClient (Apache).
- Ce module prépare le contenu et la sérialize en chaine JSON, définit l'adresse et envoie la requête HTTP.

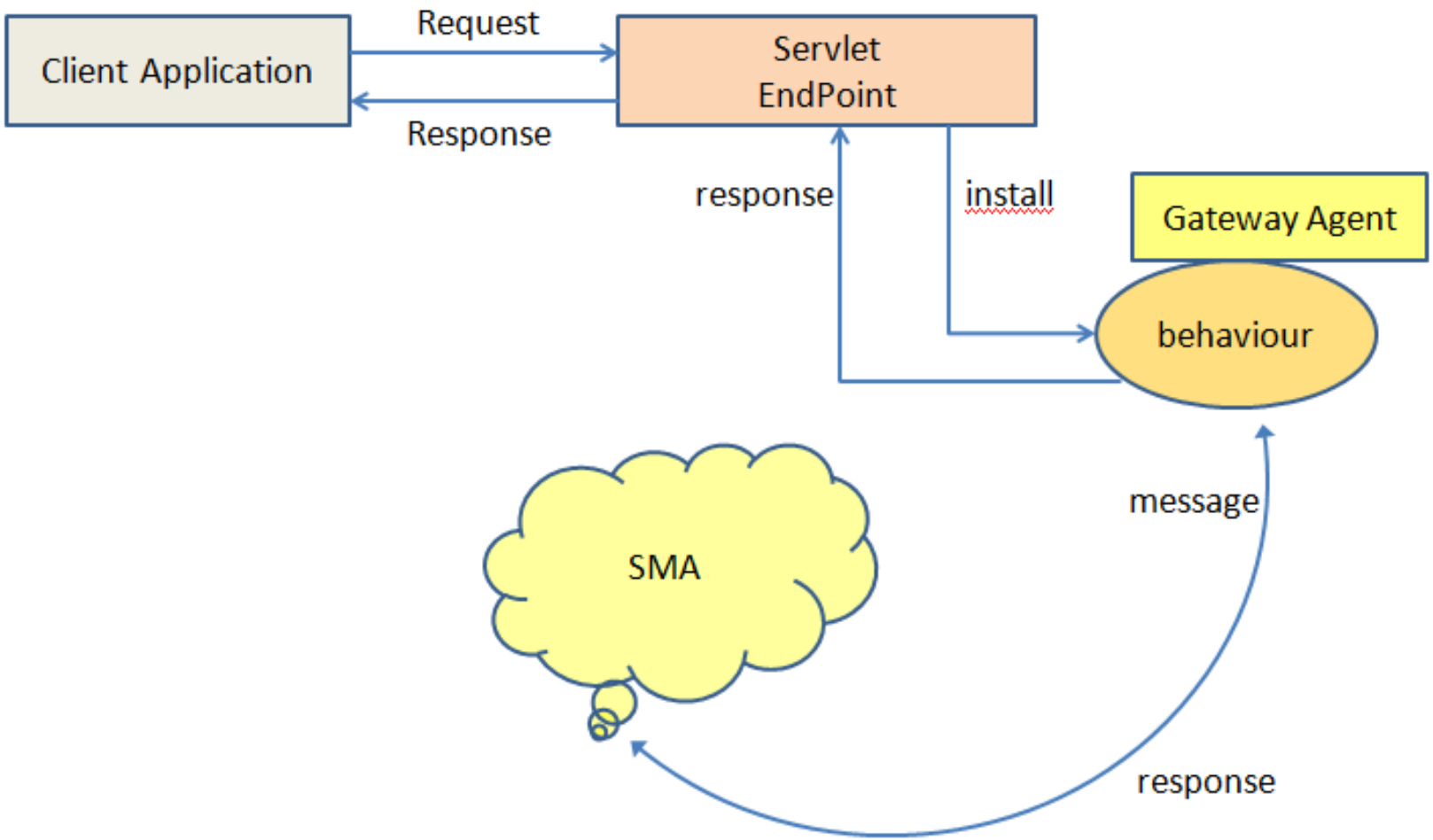
```
HttpClient client;
client = HttpClients.createDefault();
String response = postAgentMessage("/admin",content);
String response = postAgentMessage("/agent/db",content);
```

Envoi de la requête

```
public String postAgentMessage(String postAddress,
                               String content) {
 HttpPost httppost = new HttpPost(REST_URL + postAddress);
 String answer = "";
 try {
  StringEntity entity = new StringEntity(content);
  httppost.setEntity(entity);
  HttpResponse res = client.execute(httppost);
  int status = res.getStatusLine().getStatusCode();
  if (status == HttpStatus.SC_OK) {
   HttpEntity answerEntity = res.getEntity();
   answer = EntityUtils.toString(answerEntity);
  }
  else
   answer = "{\"status\" : \"Error\"}";
  } catch (IOException e) { e.printStackTrace(); }
 return answer;
}
```

Sommaire

Agent Gateway



Websockent endpoint

```
@ServerEndpoint(value = "/sma/{id}")
public class AdminAgentEndPointAnnotated {
 @OnMessage
 public String onMessage(String message,
       Session session, @PathParam("id") String id) {
  if (id == null)
     return "{\"error\" : \"no agent aid\" }";
  Map<String, Object> params = getParams(message);
  if (!test(params))
     return "{\"error\" : \"no content\" }";
  return activeAgent(id, params);
 }
...
}
```

Websockent endpoint

```
private String activeAgent(String aid,
                           Map<String, Object> params) {
 String result="";
 ACLMessage message = RequestMessage.
                      formatMessage(params);
 ProcessBehaviour behaviour =
                  new ProcessBehaviour(aid, message);
 try {
  JadeGateway.execute(behaviour);
  result = "{\"main\" : \"" + behaviour.answer + "\" }";
 } ...
 catch (InterruptedException e) {
  e.printStackTrace();
 }
 return result;
}
```

Conclusion

- Un client envoie un contenu destiné à un agent particulier
  - Un adressage différent pour chaque agent destinataire (transparent pour le client).
  - Le contenu est une Map de clés - valeurs
- Le processus réceptionneur de la requête active le Gateway.
  - Il prépare un message adapté (REQUEST, conv-id unique, contenu)
  - Il crée un behaviour et l'exécute.
  - Il attend l'arrêt du behaviour et en récupère la réponse.
  - Il retourne la réponse au client.
- Le behaviour est standard. Il installe le destinataire du message et l'envoie. Il attend un message en retour basé sur un conv-id unique.

Sommaire

Conclusion

- La base noSQL Redis possède une fonctionnalité Publish - Subscribe. Tous les clients abonnés à un channel reçoivent les messages publiés sur ce channel (même ceux publiés par eux-mêmes).

Agent Gateway