Perspectives

7.1 Synthèse des travaux réalisés

Nous avons proposé dans le cadre de cette thèse une nouvelle approche dédiée à la réalisation de jeux massivement multi-joueurs, permettant de rationaliser et de sécuriser le développement de jeux innovants en ce qui concerne la manière dont le joueur interagit avec le monde virtuel dans lequel le jeu se déroule.

Notre proposition consiste à utiliser des méthodes de génie logiciel appropriées, basées sur une démarche de prototypage et de raffinement, en utilisant un environnement de développement dédié.

Nous avons défini et prototypé un framework ouvert et modulaire pour garantir la généricité de la solution, et montré comment il peut s'intégrer dans un environnement de développement.

Ce modèle original est centré sur la définition des interactions à l'aide d'une description très fine de la manière dont les données de l'application sont répliquées sur les différents hôtes de l'application distribuée. Il adopte un modèle simple de programmation concurrente pour la description du comportement de l'application, basé sur un ordonnancement coopératif et équitable des tâches en cours d'exécution.

Plus général que les modèles orientés objet et orientés agents, il dissocie

la manière dont sont organisées les données de leurs comportement. Il permet néanmoins de concevoir une application selon ces paradigmes, en organisant de manière adéquate les données et les traitements.

Ce découplage entre les données et leur traitement est complété par une approche transverse, proche des modèles de meta-programmation tels que la programmation par aspects, où différentes manières d'effectuer un traitement peuvent être considérées lors de la conception de l'application.

7.2 Implémentation réaliste

Nous voulons dans un futur proche réaliser une implantation plus réaliste du modèle, en prenant en compte les autres aspects d'un jeu massivement multi-joueurs, dont les plus importants sont donnés ici.

Nous voulons ajouter au prototype un mécanisme permettant aux hôtes de l'application de s'échanger ou de mettre à jour dynamiquement de nouveaux comportements (les ordonnançables). Cette amélioration ne modifiera pas le modèle, mais seulement le prototype, puisque nous envisageons alors d'ajouter les comportements à la liste des types de base définissant les données propres d'un état. Un état pourra ainsi être conçu pour définir un moyen d'obtenir un mécanisme de *code mobile*.

Nous aimerions également réaliser une implantation plus performante que celle qui existe actuellement, et qui est basée sur l'ordonnancement des *threads* Java afin de réaliser un autre ordonnanceur. L'implantation actuelle utilise un *pool* de *threads* Java qui sont tous créés au lancement de l'application : la création d'un thread est très coûteuse en Java et nous devions éviter d'utiliser trop souvent cette opération pour garder un prototype un tant soit peu réaliste. Lorsqu'un réflexe est déclenché, son exécution est attribuée à un *thread* donné du *pool*. Le nombre de *threads* à créer au démarrage, ainsi que son augmentation ou diminution dynamique selon les besoins de l'application est un problème complexe. Il existe des patrons de conception répondant à cette famille de problèmes et nous comptons les étudier et expérimenter afin de trouver une solution souple et efficace. Nous n'envisageons pas forcément d'abandonner totalement le langage Java, qui peut rester le language d'implantation du reste du *framework* et de la bibliothèque, mais il est possible

que d'autres langages soient plus adéquats pour la réalisation de l'ordonnanceur lui-même, et nous comptons faire quelques expériences à ce sujet.

De même, nous comptons dans un futur très proche étendre la bibliothèque très minimale que nous proposons, notamment en incluant des éléments permettant de définir des comportements d'objets plus complexes du monde virtuel, comme par exemple des agents autonomes. Cette tâche est un bon test sur le réalisme de notre solution, dont les critères de réussite seront un bon passage à l'échelle et la simplicité des descriptions dans le cadre du *framework*.

Nous aimerions également tester l'adéquation de notre modèle à d'autres familles d'applications distribuées faisant intervenir des éléments hétérogènes et communicants disposant d'autres contraintes matérielles, comme par exemple des applications domotiques ou de robots ludiques [92]. Nous pourrions ainsi étendre de manière pertinente la bibliothèque à de nouveaux protocoles et modèles de communication qui seront peut-être, comme le laisse envisager l'intérêt actuel pour les terminaux de jeux *mobiles*, les supports des distractions de demain.

7.3 Environnement de développement

Nous voudrions dans l'idéal réaliser un environnement de développement complet autour du *framework* que nous avons défini. Celui-ci permettrait d'assembler les composants de l'application, de générer le code spécifique à l'application à partir de cet assemblage, et fournirait des outils de mise au point, comme par exemple rejouer une exécution, à l'aide de scenarii d'événements provenant du réseau ou des interfaces utilisateur.

Dans un futur proche, il sera indispensable de fournir un outil permettant de générer une grande partie du code générique, et capable d'automatiser l'écriture des scripts de configuration, ces tâches étant actuellement fastidieuses et répétitives.

Nous pensons également intégrer un outil de simulation de mauvaises conditions de réseau du style de *NetTool* [60].

Nous voulons ultérieurement étudier les possibilités de réaliser des ana-

lyses statiques s'appuyant sur la sémantique d'exécution du modèle pour identifier et calibrer les caractéristiques de l'application : par exemple, donner une estimation des ressources nécessaires au déploiement de l'application, ce qui permettrait d'éviter le sous-dimensionnement ou le sur-dimensionnement en permettant assez tôt d'avoir une estimation des ressources nécessaires.

Chapitre 8

Annexes

Nous allons présenter dans ce chapitre un complément de l'exemple décrit à la fin du chapitre 6. La première partie de ce chapitre donne la totalité du fichier de configuration utilisé côté serveur pour la réalisation de l'exemple.

La deuxième partie présente les classes Java de description des états ainsi que leurs fabriques, tandis que la troisième présente le code des ordonnançables et leurs fabriques.

Cette présentation a uniquement pour but de compléter la description de notre modélisation et d'illustrer comment le *Fill-In-The-Gaps Toolkit* fonctionne à l'heure actuelle pour le lecteur curieux.

Nous envisageons de générer tout le code présenté au sein de ce chapitre par un outil d'intégration construit autour du *Fill-In-The-Gaps Toolkit*.

8.1 Le fichier de configuration de l'application

STATE game GameStateFactory
KEYDUPLICATEDSTATESETS clients
KEYSUBSTATE lastHostCreated lastMessage newMessage murmureDistance
SUBSTATEINITIALIZE lastHostCreated lastMessage newMessage murmureDistance
BEHAVIORMODELS messageUpdate

```
DUPLICATA clients ClientFactory
KEYDATA x y name
DATAINITIALIZE name.anonymous.string
REPLICATIONMODELS forward hostCreatedReplication

STATE lastHostCreated LastHostConnectedFactory
KEYDATA hostIdentifier
DATAINITIALIZE hostIdentifier.null.string

STATE lastMessage MessageNetworkedStateFactory
KEYDATA message

STATE newMessage MessageStateFactory
KEYDATA message
REPLICATIONMODELS sendNewMessage sendMurmure

STATE murmureDistance MurmureDistanceFactory
KEYDATA distance
DATAINITIALIZE distance.100.int

REPLICATION forward StateChangedReplicationFactory
STATEOBSERVED SELF
SCOPE allButMe
PROTOCOL UDP

REPLICATION hostCreatedReplication HostConnectedReplicationFactory
STATEOBSERVED game.lastHostCreated
STATENEEDED game.lastHostCreated
SCOPE hostCreatedScope
PROTOCOL TCP

BEHAVIOR messageUpdate StateUpdateBehaviorFactory
STATEOBSERVED game.lastMessage
STATENEEDED game.lastMessage game.newMessage
REPLICATION sendNewMessage ChatReplicationFactory
STATEOBSERVED game.newMessage
STATENEEDED game.newMessage
```

SCOPE all
PROTOCOL TCP

REPLICATION sendMurmure MurmureReplicationFactory
STATEOBSERVED game.newMessage
STATENEEDED game.newMessage game.murmureDistance game.clients
SCOPE murmureScope
PROTOCOL TCP

SCOPEDEF allButMe AllButMeScopeFactory

SCOPEDEF all AllScopeFactory

SCOPEDEF hostCreatedScope HostConnectedScopeFactory

SCOPEDEF murmureScope MurmureScopeFactory

PROTOCOLDEF TCP TCPProtocolFactory

PROTOCOLDEF UDP UDPProtocolFactory

8.2 Classes et fabriques d'états

8.2.1 L'état gameState

8.2.1.1 Classe GameState

```
public class GameState extends State {
  private DuplicateStatesSet clients;
  private LastHostConnected lastHostConnected;
  private MessageNetworkedState lastMessage;
  private MessageState newMessage;
  private MurmureDistance murmureDistance;

  public GameState(String path, DataAccessList dataAccessList,
```

```
    String subStatesKeys, StateAccessList substateSetters,
    String datasInitialization, String replicationKeys,
    String behaviorKeys, String substateInitialization,
    String keyDuplicatedStates,
    StateAccessList duplicatedStatesInitializers,
    StateFactory factory) {
  super(path, dataAccessList, subStatesKeys, substateSetters,
      datasInitialization, replicationKeys, behaviorKeys,
      substateInitialization, keyDuplicatedStates,
      duplicatedStatesInitializers, factory);
}
public DuplicateStatesSet getClients() {
  return clients;
}
public void setClients(DuplicateStatesSet clients) {
  this.clients = clients;
}
public synchronized LastHostConnected getLastHostConnected() {
  return lastHostConnected;
}
public synchronized void setLastHostConnected(
    LastHostConnected lastHostConnected) {
  this.lastHostConnected = lastHostConnected;
}
public synchronized MessageNetworkedState getLastMessageState() {
  return lastMessage;
}
public synchronized void setLastMessageState(
    MessageNetworkedState lastMessageState) {
  this.lastMessage = lastMessageState;
}
public synchronized MessageState getNewMessage() {
  return newMessage;
}
public synchronized void setNewMessage(MessageState newMessage) {
  this.newMessage = newMessage;
}
public synchronized MurmureDistance getMurmureDistance() {
  return murmureDistance;
}
```

```
  public synchronized void setMurmureDistance(
      MurmureDistance murmureDistance) {
    this.murmureDistance = murmureDistance;
  }
}
```

8.2.1.2 Classe GameStateFactory

```
public class GameStateFactory extends StateFactory {
  public State createEmptyState() {
    return new GameState(null, null, null, null, null, null, null,
        null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    String subStateKeys = subStateDependancies(key, keySubstate);
    StringTokenizer keySubstateTokenizer = new StringTokenizer(
        subStateKeys, " ");
    String lastHostKey = keySubstateTokenizer.nextToken();
    String lastMessageKey = keySubstateTokenizer.nextToken();
    String newMessageKey = keySubstateTokenizer.nextToken();
    String murmureDistanceKey = keySubstateTokenizer.nextToken();
    StateAccessList stateSetterList = new StateAccessList();
    stateSetterList.put(lastHostKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getLastHostConnected();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setLastHostConnected((LastHostConnected) state);
      }
    });
    stateSetterList.put(lastMessageKey, new StateAccessInState() {
```

```
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getLastMessageState();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setLastMessageState((MessageNetworkedState) state);
      }
    });
    stateSetterList.put(newMessageKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver).getNewMessage();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setNewMessage((MessageState) state);
      }
    });
    stateSetterList.put(murmureDistanceKey, new StateAccessInState() {
      public StateComponent getState(
          StateCreationObserver stateCreationObserver) {
        return ((GameState) stateCreationObserver)
            .getMurmureDistance();
      }
      public void setState(
          StateCreationObserver stateCreationObserver,
          StateComponent state) {
        ((GameState) stateCreationObserver)
            .setMurmureDistance((MurmureDistance) state);
      }
    });
    String duplicatedStates = subStateDependancies(key,
        keyDuplicatedStates);
    StringTokenizer st = new StringTokenizer(duplicatedStates, " ");
```

```
    String clientKey = st.nextToken();
    StateAccessList duplicatedStatesInitialization = new StateAccessList();
    duplicatedStatesInitialization.put(clientKey,
        new StateAccessInState() {
          public StateComponent getState(
              StateCreationObserver stateCreationObserver) {
            return ((GameState) stateCreationObserver).getClients();
          }
          public void setState(
              StateCreationObserver stateCreationObserver,
              StateComponent state) {
            ((GameState) stateCreationObserver)
                .setClients((DuplicateStatesSet) state);
          }
        });
    return new GameState(key, null, subStateKeys, stateSetterList,
        null, replication, behaviorKeys, subStateDependancies(key,
            substateInitialization), duplicatedStates,
        duplicatedStatesInitialization, this);
  }
}
```

8.2.2 Les états duplicata clients

8.2.2.1 Classe Client

```
public class Client extends NetworkState {
  public final static int RAY = 10;
  private int x;
  private int y;
  private String name;

  public Client(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
```

```
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized int getX() {
    return x;
  }
  public synchronized void setX(int x) {
    this.x = x;
  }
  public synchronized int getY() {
    return y;
  }
  public synchronized void setY(int y) {
    this.y = y;
  }
  public synchronized String getName() {
    return name;
  }
  public synchronized void setName(String name) {
    this.name = name;
  }
}
```

8.2.2.2 Classe ClientFactory

```
public class ClientFactory extends StateFactory {
  public State createEmptyState() {
    return new Client(null, null, null, null, null, null, null, null,
        null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String xKey = keyDataTokenizer.nextToken();
```

```
    String yKey = keyDataTokenizer.nextToken();
    String nameKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(xKey, new DataAccess(DataAccess.INTEGER_TYPE) {
      protected void setData(State state, int value) {
        ((Client) state).setX(value);
      }
      protected int getInt(State state) {
        return ((Client) state).getX();
      }
    });
    dataSetterList.put(yKey, new DataAccess(DataAccess.INTEGER_TYPE) {
      protected void setData(State state, int value) {
        ((Client) state).setY(value);
      }
      protected int getInt(State state) {
        return ((Client) state).getY();
      }
    });
    dataSetterList.put(nameKey,
        new DataAccess(DataAccess.STRING_TYPE) {
          protected void setData(State state, String value) {
            ((Client) state).setName(value);
          }
          protected String getString(State state) {
            return ((Client) state).getName();
          }
        });
    return new Client(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys,
        substateInitialization, null, null, this);
  }
}
```

8.2.3 L'état lastMessage

8.2.3.1 Classe MessageNetworkedState

```
public class MessageNetworkedState extends NetworkState {
```

```
  private String message;

  public MessageNetworkedState(String path,
      DataAccessList dataAccessList, String subStatesKeys,
      StateAccessList substateSetters, String datasInitialization,
      String replicationKeys, String behaviorKeys,
      String substateInitialization, String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getMessage() {
    return message;
  }
  public synchronized void setMessage(String message) {
    this.message = message;
  }
}
```

8.2.3.2 Classe MessageNetworkedStateFactory

```
public class MessageNetworkedStateFactory extends StateFactory {
  public State createEmptyState() {
    return new MessageNetworkedState(null, null, null, null, null,
        null, null, null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String messageKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(messageKey, new DataAccess(
        DataAccess.STRING_TYPE) {
```

```
      protected void setData(State state, String value) {
        ((MessageNetworkedState) state).setMessage(value);
      }
      protected String getString(State state) {
        return ((MessageNetworkedState) state).getMessage();
      }
    });
    return new MessageNetworkedState(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys, null, null, null,
        this);
  }
}
```

8.2.4 L'état newMessage

8.2.4.1 Classe MessageState

```
public class MessageState extends State {
  private String message;

  public MessageState(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getMessage() {
    return message;
  }
  public synchronized void setMessage(String message) {
    this.message = message;
  }
}
```

8.2.4.2 Classe MessageStateFactory

```
public class MessageStateFactory extends StateFactory {
  public State createEmptyState() {
    return new MessageState(null, null, null, null, null, null, null,
        null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    String messageKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(messageKey, new DataAccess(
        DataAccess.STRING_TYPE) {
      protected void setData(State state, String value) {
        ((MessageState) state).setMessage(value);
      }
      protected String getString(State state) {
        return ((MessageState) state).getMessage();
      }
    });
    return new MessageState(key, dataSetterList, null, null,
        dataInitialize, replication, behaviorKeys, null, null, null,
        this);
  }
}
```

8.2.5 L'état lastHostConnected

8.2.5.1 Classe LastHostConnected

```
public class LastHostConnected extends State {
  private String hostIdentifier;

  public LastHostConnected(String path,
      DataAccessList dataAccessList, String subStatesKeys,
```

```
      StateAccessList substateSetters, String datasInitialization,
      String replicationKeys, String behaviorKeys,
      String substateInitialization, String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
  }
  public synchronized String getHostIdentifier() {
    return hostIdentifier;
  }
  public synchronized void setHostIdentifier(String hostIdentifier) {
    this.hostIdentifier = hostIdentifier;
  }
}
```

8.2.5.2 Classe LastHostConnectedFactory

```
public class LastHostConnectedFactory extends StateFactory {
  public State createEmptyState() {
    return new LastHostConnected(null, null, null, null, null, null,
        null, null, null, null, this);
  }
  public State createState(String key, String keyData,
      String dataInitialize, String replication, String behaviorKeys,
      String keySubstate, String substateInitialization,
      String keyDuplicatedStates) {
    StringTokenizer keyDataTokenizer = new StringTokenizer(keyData,
        " ");
    // définition des initialisateurs d'attributs de types simples(non états)
    String hostIdentifierKey = keyDataTokenizer.nextToken();
    DataAccessList dataSetterList = new DataAccessList();
    dataSetterList.put(hostIdentifierKey, new DataAccess(
        DataAccess.STRING_TYPE) {
      protected void setData(State state, String value) {
        ((LastHostConnected) state).setHostIdentifier(value);
      }
```

```
      protected String getString(State state) {
        return ((LastHostConnected) state).getHostIdentifier();
      }
    });
    return new LastHostConnected(key, dataSetterList, null, null,
        dataInitialize, null, null, null, null, null, this);
  }
}
```

8.2.6 L'état murmureDistance

8.2.6.1 Classe MurmureDistance

```
public class MurmureDistance extends State {
  private int distance;

  public MurmureDistance(String path, DataAccessList dataAccessList,
      String subStatesKeys, StateAccessList substateSetters,
      String datasInitialization, String replicationKeys,
      String behaviorKeys, String substateInitialization,
      String keyDuplicatedStates,
      StateAccessList duplicatedStatesInitializers,
      StateFactory factory) {
    super(path, dataAccessList, subStatesKeys, substateSetters,
        datasInitialization, replicationKeys, behaviorKeys,
        substateInitialization, keyDuplicatedStates,
        duplicatedStatesInitializers, factory);
    // TODO Auto-generated constructor stub
  }
  public synchronized int getDistance() {
    return distance;
  }
  public synchronized void setDistance(int distance) {
    this.distance = distance;
  }
}
```