ESPRITSYSTEMS

Remote Property Binding

Konzept und Anwendung von Remote-Properties in der Esprit Client/Server Technologie

Oktober 2016
Rainer Büsch

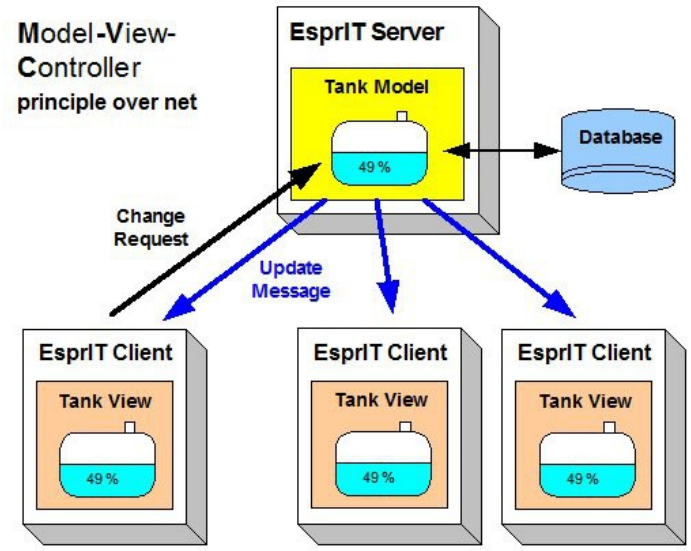
Inhaltsverzeichnis

1 Einführung

Dieses Dokument beschreibt das Prinzip und die Funktionsweise der *Remote-Properties*, wie sie in der *Esprit Client/Server Technologie* verwendet werden. Es erläutert anhand von Programmierbeispielen den Umgang mit ihnen und zeigt, wie sie effizient eingesetzt werden können, um komplexere Datenmodelle aufzubauen. Die folgenden Ausführungen erheben keinen Anspruch auf Vollständigkeit. Vielmehr geht es darum, dem Leser ein Grundverständnis zu vermitteln und ihm eine Übersicht über die Möglichkeiten der *Remote-Properties* zu geben.

1.1 Model-View-Controller (MVC)

Durch seine besondere Architektur erlaubt der Esprit-Server die vollständige Realisierung des *Model-View-Controller* Prinzips über Netz. Bei diesem Prinzip existiert ein Zentrales Datenmodell auf dem Server (siehe *TankModel* im untenstehenden Bild). Wird dieses durch einen Client-Request geändert, dann werden Aktualisierungs-Events an alle anderen Clients verschickt, so dass diese den neuen Zustand unmittelbar reflektieren können. Für die erforderlichen Requests müssen typischerweise spezielle *Agenten* geschrieben werden. Ebenso müssen für die erforderlichen Aktualisierungen spezielle *Events* formuliert werden.



Model-View-Controller Prinzip über Netz

2 Remote-Properties

Die **Remote-Properties** stellen eine generalisierte Anwendung des MVC Prinzips dar, bei der es nicht mehr notwendig ist, eigene *Agenten* oder *Events* zu entwickeln. Remote-Properties bestehen aus zwei Komponenten:

➔ **ServerProperty** (Interface)
Diese existiert auf der Server-Seite und kapselt einen beliebigen Wert, welcher von Clients zugegriffen und verändert werden kann. Änderungen werden per Event an die Clients notifiziert.

➔ **ClientProperty** (Interface)
Diese existiert auf der Client-Seite und bildet das Pendant zu einer bestimmten *ServerProperty*. Bei ihrer Initialisierung übernimmt sie den Wert der zugehörigen *ServerProperty* und aktualisiert sich von da an selbständig bei künftigen Veränderungen.

Sowohl *ServerProperty* als auch *ClientProperty* implementieren das Interface *RemoteProperty*, welches die Methode *getPropertyName()* vorschreibt. Der vergebene Property-Name muss eindeutig sein. Er allein stellt den Bezug zwischen einer *ServerProperty* und einer *ClientProperty* her.

2.1 Server-Property

Für die Serverseite gibt zur Zeit folgende Implementierungen von *ServerProperty*:

➔ **ServerValueProperty<T>**
Diese verwaltet ein beliebiges, generisches Objekt *T* als Wert. Zur Änderung steht die Methode *setValue(SessionId, T)* zur Verfügung. Bei Änderungen des Wertes werden die Clients notifiziert.

➔ **ServerListProperty<T>**
Diese verwaltet eine *List<T>* von beliebigem Typ *T*. Zur Veränderung der Liste stehen typische Listen-Methoden zur Verfügung, wie *add(SessionId,T), remove(SessionId,T)* und *clear(SessionId)*. Ändert sich ein Wert in der Liste, so wird nicht die ganze Liste, sondern nur die Änderung an die Clients notifiziert.

➔ **OrderedServerListProperty<T>**
Ist eine Erweiterung von **ServerListProperty**, die zusätzlich Methoden zur Index-orientierten Änderung der Liste bereitstellt: *insertAt(int,T), replaceAt(SessionId,int,T), removeAt(SessionId,int)*. Desweiteren gibt es die Methoden, um die Position eines Wertes in der Liste zu ändern: *moveTo(SessionId,int,T), moveUp(SessionId,T) und moveDown(SessionId,T).*

➔ **ServerMapProperty<K,T>**
Diese verwaltet eine *Map<K,T>* von beliebigem Key *K* und Value *T*. Zur Veränderung der Map stehen typische Map-Methoden zur Verfügung, wie *put(SessionId,K,T), remove(SessionId,K)* und *clear(SessionId)*. Ändert sich ein Wert in der Map, so wird nicht die ganze Map, sondern nur die Änderung an die Clients notifiziert.

Man beachte, dass alle Methoden, die eine *ServerProperty* verändern, eine *SessionId* als Argument verlangen! Daher ist serverseitig stets bekannt, wer wann die letzte Änderung vorgenommen hat. Dies wird auch den Clients per Event mitgeteilt.

➔ Die Werte-Objekte <T> einer *ServerProperty* müssen serialisierbar sein (sprich: das *Serializable* Interface implementieren), damit sie zwischen Client und Server ausgetauscht werden können.

2.1.1 Erstellung einer ServerProperty

Als Beispiel für eine einfache *ServerValueProperty* wollen wir uns die *ServerPositionProperty* anschauen. Diese verwaltet eine x/y-Position in Form eines einfachen *Point* Objekts. Sie sieht folgendermaßen aus:

```
public final class ServerPositionProperty extends ServerValueProperty<Point> {
  private final Dimension range = new Dimension(500, 500);

  public ServerPositionProperty(ServerContext ctx) {
     super(ctx, "global-position");
  }
```

```
    @Override
    public void init() throws Exception {
        super.init();
        Point intialValue = new Point(range.width/2, range.height/2);
        setValue(getServerContext().getServerSessionId(), intialValue));
    }

    protected Point checkInputValue(SessionId sessionId, Point newValue) throws Exception {
        if (newValue.getX() > range.width) {
            throw new IllegalArgumentException("x-value out of range");
        }

        if (newValue.getY() > range.height) {
            throw new IllegalArgumentException("y-value out of range");
        }
        return newValue;
    }
}
```

Wie man sieht, leitet die *ServerPositionProperty* von der generischen *ServerValueProperty<T>* ab und legt *Point* als Wertetyp fest. Außerdem wird im Konstruktor ein **eindeutiger Name** festgelegt, der auch den Clients bekannt sein muss, damit eine Verbindung hergestellt werden kann.

Wird nun mit *setValue(SessionId, Point)* der Wert verändert, so wird implizit auch die optional überschriebene Methode *checkInputValue(SessionId, Point)* aufgerufen. Im obigen Beispiel wird diese verwendet, um den Wertebereich zu überprüfen. Bei einem ungültigen Wert würde der Versuch für den betreffenden Client mit einer Fehlermeldung enden.

2.1.2 Instantiierung einer Server-Property

Eine *ServerProperty* benötigt zur Instantiierung die Laufzeitumgebung eines Esprit-Servers. Typischerweise wird sie im Konstruktor ihres speziellen *ServerContext*'s erzeugt und in der *serverInit()* Methode initialisiert.

```
    ServerPositionProperty serverPosition = new ServerPositionProperty(this); // ServerContext
    … … …
    @Override
    protected void serverInit() throws Exception {
        serverPosition.init();
    }
```

Bei der Initialisierung setzt sie sich selbst ihren initialen Wert. Außerdem wird sie beim *ServerPropertyManager* zur Verwaltung eingetragen. Dieser stellt die Eindeutigkeit ihres Namens sicher und managed ihren Lebenszyklus. Ab diesem Moment steht die Property für Client-Zugriffe zur Verfügung.

2.2 Client-Property

Für die Clientseite gibt es folgende Implementieren von *ClientProperty*:

- **ClientValueProperty<T>** (GUI-neutral)
  FXClientValueProperty<T> (für JavaFX, abgeleitet von *ObjectPropertyBase*)
  Diese verwaltet ein beliebiges, generisches Objekt *T* als Wert. Zur Änderung steht die Methode *setValue(T)* zur Verfügung, die den Aufruf über Netz an die korrespondierende *ServerValueProperty<T>* delegiert.
- **ClientListProperty<T>**
  Diese verwaltet eine *List<T>* von beliebigem Typ *T*. Die Liste kann lediglich ausgelesen werden. Es gibt keine Methoden zur Veränderung.

➔ **ModifiableClientListProperty<T>**
Erweitert *ClientListProperty* um typische Listen-Methoden zur Veränderung, wie *add(T), remove(T)* und *clear()*. Alle Methoden delegieren den Aufruf jeweils an die korrespondierende *ServerListProperty<T>*.

➔ **OrderedClientListProperty<T>**
Erweitert *ModifiableClientListProperty* um typische Methoden zur index-orientierten Veränderung, wie *insertAt(int, T), replaceAt(int, T), removeAt(int)*. Zusätzlich gibt es Methoden, um die Position eines Wertes in der Liste zu ändern: *moveTo(int,T) moveUp(T) und moveDown(T)*.

➔ **ClientMapProperty<K,T>**
Diese verwaltet eine *Map<K,T>* von beliebigen Key *K* und Value *T* Paaren. Die Map kann lediglich ausgelesen werden. Es gibt keine Methoden zur Veränderung.

➔ **ModifiableClientMapProperty<K,T>**
Erweitert *ClientMapProperty* um Methoden zur Veränderung wie *put(K,T), remove(K)* und *clear()*. Alle Methoden delegieren den Aufruf jeweils an die korrespondierende *ServerMapProperty<K,T>*.

2.2.1 Erstellung einer ClientProperty

Als Beispiel für eine einfache *ClientValueProperty* wollen wir uns als Gegenstück zur *ServerPositionProperty* die *ClientPositionProperty* anschauen, die folgendermaßen formuliert ist:

```
public final class FXClientPositionProperty extends FXClientValueProperty<Point> {

  public FXClientPositionProperty(ClientContext ctx) {
     super(ctx, "global-position");
  }
}
```

Wie man sieht, leitet die *FXClientPositionProperty* von der generischen Klasse *FXClientValueProperty<T>* ab und legt den Typ sowie den Namen fest.

➔ Property-Name und -Typ müssen selbstverständlich mit der serverseitigen *ServerPositionProperty* übereinstimmen!

2.2.2 Instantiierung einer ClientProperty

Zur Instantiierung benötigt die *ClientProperty* die Laufzeitumgebung eines Clients, also einen *ClientContext*. Die Anwendung sieht dann wie folgt aus:

```
FXClientPositionProperty positionProp = new FXClientPositionProperty(ctx); // ClientContext

positionProp.goOnline();
```

Der Aufruf der *goOnline()* Methode bewirkt, dass der initiale Wert vom Server geholt wird. Ab diesem Moment ist die *ClientProperty* online und aktualisiert sich selbständig bei Änderungen.

2.2.3 Property-Binding an JavaFX-Komponenten

Man beachte, dass *FXClientValueProperty<T>* von der JavaFX-Klasse *ObjectPropertyBase<T>* ableitet und damit eine gewöhnliche *JavaFX* Property darstellt, die mit den *JavaFX* typischen Binding-Mechanismen an entsprechende GUI-Komponenten gebunden werden kann! Das Binding kann bidirektional sein. D.h.: bei Änderung des Wertes der *ClientProperty* wird auch die betreffende *ServerProperty* geändert, was wiederum dazu führt, dass **alle** Clients ihre lokale Property aktualisieren.

2.2.4 Property-Binding an Swing-Komponenten

Für die Swing-Welt existieren völlig analoge Implementierungen für die clientseitigen Properties. Da es in Swing kein Property-Binding wie in *JavaFX* gibt, muss man sich hier für die remoten Events bei der Property direkt zu registrieren, wie im folgenden Beispiel gezeigt:

```
ClientPositionProperty positionProp = new ClientPositionProperty(this); // ClientContext

positionProp.init();
positionProp.addRemoteValueChangeListener(e → remoteChangeReceived(e));

private void remoteChangeReceived(RemoteValueChangeEvent e) {
  System.out.println("Update received: "+e);
 // here you may update any GUI-component
}
```

3 Datenmodelle mit Remote-Properties

3.1 Server-Model mit Remote-Properties

Remote-Properties eignen sich hervorragend als Bausteine zum Aufbau komplexerer Datenmodelle. Dies soll am Beispiel des *ColorServerModel's* gezeigt werden, welches drei Integer-Properties für die drei Farbkomponenten rot, grün und blau kombiniert:

```
public class ColorServerModel extends AbstractServerComponent<ServerContext> {
  private final ServerValueProperty<Integer> colorRed;
  private final ServerValueProperty<Integer> colorGreen;
  private final ServerValueProperty<Integer> colorBlue;

  public ColorServerModel(ServerContext ctx) {
     super(ctx);
     colorRed = new ServerValueProperty<>(ctx, "server.color.red", 0);
     colorGreen = new ServerValueProperty<>(ctx, "server.color.green", 0);
     colorBlue = new ServerValueProperty<>(ctx, "server.color.blue", 0);
  }

  @Override
  public void init() throws Exception {
     super.init();
     colorRed.init(); colorGreen.init(); colorBlue.init();
  }
}
```

Dieses *ColorServerModel* wird auf die übliche Art und Weise im *ServerContext* eingebaut und ist nach der Initialisierung unmittelbar funktionsfähig. Man beachte, dass es keinerlei Zugriffsmethoden besitzt, da sämtliche Abfragen und Änderungen transparent über die *RemoteProperty*-API abgehandelt werden.

3.2 Client-View-Model mit Remote-Properties

Das zum *ColorServerModel* zugehörige Client-View-Model sieht dann folgendermaßen aus:

```
public final class FXColorViewModel extends AbstractClientObject<ClientContext> {

  private final FXClientValueProperty<Number> propertyRed;
  private final FXClientValueProperty<Number> propertyGreen;
  private final FXClientValueProperty<Number> propertyBlue;

  private final SimpleStringProperty propertyLastAccessor;
  private final SimpleObjectProperty<Color> propertyColor;
```

```
  public FXColorViewModel(ClientContext ctx) {
     super(ctx);
     propertyRed = new FXClientValueProperty<>(ctx, "server.color.red");
     propertyGreen = new FXClientValueProperty<>(ctx, "server.color.green");
     propertyBlue = new FXClientValueProperty<>(ctx, "server.color.blue");

     propertyColor = new SimpleObjectProperty<>();
     propertyLastAccessor = new SimpleStringProperty();

     propertyRed.addListener((o, ov, nv) -> colorChangedBy(propertyRed.getLastChangedBy()));
     propertyGreen.addListener((o, ov, nv) -> colorChangedBy(propertyGreen.getLastChangedBy()));
     propertyBlue.addListener((o, ov, nv) -> colorChangedBy(propertyBlue.getLastChangedBy()));
  }

  public final FXClientValueProperty<Number> propertyRed() {
     return propertyRed;
  }

  public final FXClientValueProperty<Number> propertyGreen() {
     return propertyGreen;
  }

  public final FXClientValueProperty<Number> propertyBlue() {
     return propertyBlue;
  }

  public final SimpleStringProperty propertyLastAccessor() {
     return propertyLastAccessor;
  }

  public final ObjectProperty<Color> propertyColor() {
     return propertyColor;
  }

  public void init() {
     propertyRed.goOnline();
     propertyGreen.goOnline();
     propertyBlue.goOnline();
  }

  private void colorChangedBy(SessionId sessionId) {
     Number r = propertyRed.get();
     Number g = propertyGreen.get();
     Number b = propertyBlue.get();
     if (red != null && blue != null && green != null) {
        java.awt.Color awtColor = new java.awt.Color(r.intValue(), g.intValue(), b.intValue());
        propertyColor.set(FXUtil.toFXColor(awtColor)); // convert AWT-Color to FX-Color
        propertyLastAccessor.set(sessionId.getUserInfo());
     }
  }
}
```

Für die drei Farbkomponenten wird jeweils eine *FXClientValueProperty* instantiiert, die jeweils in Namen und Typ mit der korrespondierenden *ServerValueProperty* übereinstimmen muss. In der *init()* Methode holen sich die Properties ihren Anfangswert vom Server und horchen fortan auf Aktualisierungs-Events. Neben den Farb-Properties werden praktischerweise noch zwei weitere Hilfsproperties zur Verfügung gestellt:

- ➔ eine Color-Property
  Diese kombiniert die drei Farbkomponenten zu einem *Color*-Objekt.
- ➔ eine Last-Accessor-Property
  Diese enthält die Benutzerinformation der letzten Änderung als *String*.

Diese Hilfsproperties sind als Listener bei den Farb-Properties registriert und aktualisieren sich daher automatisch bei jeder Farb-Änderung. Im Endeffekt stellt dieses View-Model seine gesamte Funktionalität über fünf Properties zur Verfügung. Der folgende Code-Ausschnitt zeigt, wie die GUI-Komponenten an die Properties des View-Models an gebunden werden:

```
  FXColorViewModel colorModel = new FXColorViewModel(contr.getContext());
  colorModel.init(); // fetches the initial server values
```

```
Slider redSlider = new Slider() {
redSlider.valueProperty().bindBidirectional(colorModel.propertyRed());
...
Label accessLabel = new Label();
accessLabel.textProperty().bind(colorModel.propertyLastAccess());
...
```

3.3 Vorteile des View-Models

Das oben gezeigte *FXColorViewModel* besteht ausschließlich aus Properties. Es ist völlig frei von GUI-Komponenten und daher isoliert zu 100% testbar. Die Anbindung an GUI-Komponenten geschieht ausschließlich über die Binding-Mechanismen von *JavaFX*, die per se bereits getestet sind und zuverlässig funktionieren. Die Netzwerkzugriffe der Remote-Properties sind für den Programmierer völlig transparent.

4 Remote-List-Properties

Wir haben gesehen, wie mit Hilfe von *ServerValueProperties* bestimmte Objekte zentral zur Verfügung gestellt werden können. Bei jeder Änderung des Objektes wird dieses serialisiert und in einem Aktualisierungs-Event an alle Clients geschickt. Dies ist völlig unproblematisch, solange das Objekt „klein“ ist. Besteht das Objekt allerdings aus einer im Prinzip beliebig langen Liste, dann kann diese Technik ineffizient werden.

Aus diesem Grunde gibt es die *ServerListProperty* zur zentralen Verwaltung von Listen eines beliebigen Typs. Ihr Gegenstück ist die *ClientListProperty,* die bei ihrer Initialisierung einmalig eine clientseitige Kopie der Liste anlegt. Alle Änderungen an der Liste erfolgen fortan im Delta-Verfahren – es wird also nicht die ganze Liste, sondern nur die letzte Änderung serialisiert. Für Remote-List-Properties gibt es folgende Implementierungen:

- **Serveseitig**
  GenericServerListProperty<T>
  Verwaltet eine Liste von Objekten eines beliebigen Datentyps.

  SessionServerListProperty<T extends SessionObject>
  Verwaltet eine Liste von *SessionObject* Instanzen, die einem Benutzer „gehören“.
  Jeder Benutzer kann Objekte dort einstellen, sieht aber jeweils nur die, der er selbst eingestellt hat.

- **Clientseitig**
  ClientListProperty<T> (read-only)
  ClientModifiableListProperty<T> (read-modify)

Zur Veränderung der Liste stehen in der *ClientModifiableListProperty* typische Listen-Verwaltungsmethoden zur Verfügung, wie *add(T), insertAt(int,T), replaceAt(int,T), remove(int,T), removeAt(int)* und *clear()*. Zusätzlich weitere Methoden, um Werte in der Liste zu verschieben, wie *moveUp(T)*, *moveDown(T)* und *moveTo(int,T)*. Die entsprechenden Methoden der *ServerListProperty* verlangen jeweils noch zusätzlich das *SessionId* Argument.

4.1 ServerModel mit ServerListProperty

Als Beispiel für die Verwendung von remoten List-Properties entwickeln wir ein einfaches *CheckListServerModel*, welches eine String-Liste zur zentralen Verwaltung von Aufgaben zur Verfügung stellt:

```
public class CheckListServerModel extends AbstractServerComponent<ServerContext> {
  private final ServerListProperty<String> propCheckList;

  public CheckListServerModel(ServerContext serverCtx) {
     super(serverCtx);
     propCheckList = new GenericServerListProperty<>(serverCtx, "server.checklist");
  }

  @Override
  public void init() throws Exception {
     super.init();
     propCheckList.init(); // makes the property available to clients
  }
  @Override
  public void destroy() throws Exception {
     super.destroy();
     propCheckList.destroy(); // makes the property unavailable to clients
  }
}
```

4.1.1 Listen-Operationen

Im laufenden Betrieb des Esprit-Servers können auf einer *ServerListProperty* im Prinzip viele Clients (unterschiedliche Session-Threads) gleichzeitig Operationen ausführen. Daher sind alle ihre Methoden als *synchronized* deklariert. Trotzdem muss bei der Anwendung sehr auf korrekte Synchronisation geachtet werden. Das folgende folgendes Beispiel zeigt eine problematische Anwendung:

```
ServerListProperty<String> prop = new ServerListProperty<>(serverCtx, "checkList");
prop.init();

List<String> hits = new ArrayList();
for (String text : prop.getList()) {
  if (text.startsWith("A")) {
     hits.add(text);
  }
}
```

Hier werden bestimmte Objekte gesucht und und in einer Hitliste aufgesammelt. Diese Iteration ist allerdings eine **unsichere Operation**, da die Liste sich während der Iteration aufgrund anderer Client-Aktivität ändern kann. Die *ServerListProperty* bietet eine Reihe synchronisierter Methoden an, mit denen **sichere Operationen** möglich sind. Ein Beispiel ist die ***find(ValueFilter)*** Methode, mit der das gleiche Ergebnis folgendermaßen erreicht wird:

```
ValueFilter filter = e → e.startsWith("A");
List<String> hits  prop.find(filter)
```

Sie *ServerListProperty* besitzt zwei Methoden, um an die verwaltete Liste zu gelangen:

- **getList()**
  gibt die Referenz auf die verwaltete Liste heraus.
  Änderungen direkt in dieser Liste sind also möglich und in speziellen Fällen auch notwendig.
  Dies sollte allerdings nur verwendet werden, wenn die Property nicht im Client-Zugriff ist.
- **getListCopy()**
  gibt eine Kopie der verwalteten Liste heraus, die dann Thread-safe durchsucht werden kann.
  Änderungen in dieser Liste haben allerdings keine Auswirkung auf die tatsächlich verwaltete Liste.

4.1.2 Methoden zur Listen-Manipulation

Eine *ServerListProperty* enthält folgende allgemeine Methoden zur Listen-Manipulation:

➔ ***add(SessionId, T)***
fügt einen neuen Wert in die Liste ein.

➔ ***update(SessionId, T)***
aktualisiert einen Wert in der Liste. Wichtig ist, dass der betreffende Wert in der Liste per *equals(Object)* Methode gefunden wird, um ihn aktualisieren zu können. Diese Art der Aktualiserung ist nur geeignet für Objekte, die einen festen „Primary Key“ besitzen.

➔ ***replace(SessionId, T_{old}, T_{new})***
aktualisiert einen Wert in der Liste. Hierbei wird der alte Wert in der Liste gesucht und durch den Neuen ersetzt. Ein „Primary Key“ ist hierbei nicht erforderlich.

➔ ***clear(SessionId)***
löscht den Inhalt der Liste.

➔ Man beachte, dass keine Methoden existieren, die einen Listenwert per Index ändern. Diese stehen erst in der Subklasse *OrderedServerListProperty* zur Verfügung.

Eine *GenericServerListProperty* bietet zusätzlich weitere Methoden zur Listen-Manipulation, die die Werte einer Liste index-bezogen ändern, wie z.B. ***insertAt(...), replaceAt(...) moveTo(...)*** etc..

4.1.3 Weitere Einstellungen

Mit ***setComparator(Comparator<T>)*** wird eine *ServerListProperty* so eingestellt, dass die Werte gemäß dem Comparator sortiert werden.

Mit der Einstellung ***setUnique(true)*** wird festgelegt, dass keine Doppeleinträge erlaubt sind.

Mit der Einstellung ***setReadOnly(true)*** kann die Property von Clients nur gelesen, aber nicht mehr verändert werden. Diese Einstellung kann jederzeit im laufenden Betrieb geändert werden.

4.2 Client View-Model mit ClientListProperty

Das *FXCheckListViewModel* der Clientseite enthält die entsprechende *ClientListProperty* von gleichem Namen und Typ, wie die *ServerListProperty*. Der Java-Code sieht wie folgt aus:

```
public final class FXCheckListViewModel extends AbstractClientObject<ClientContext> {
  private final ClientListProperty<String> remoteCheckList;

  public FXCheckListViewModel(ClientContext ctx) {
     super(ctx);
     remoteCheckList = new ClientListProperty<>(ctx, "server.checklist");
     remoteCheckList.setList(new ObservableArrayList<>()); // special for FX-GUI
  }

  public final<String> remoteCheckList() {
     return remoteCheckList;
  }

  public final ObservableList<String> propertyCheckList() {
     return (ObservableList<String>)remoteCheckList.getList();
  }

  public void init() {
     remoteCheckList.goOnline();
  }
}
```

Die *ClientListProperty* enthält eine *ObservableList* Instanz, die mit *propertyCheckList()* nach außen zur Verfügung gestellt wird. Diese kann direkt z.B. an eine *ListView* GUI-Komponente gebunden werden.

Zur Modifikation der Liste müssen die speziellen Listen-Verwaltungsmethoden der *ClientListProperty* benutzt werden. Dazu wird deren Referenz mit *remoteCheckList()* nach außen

gereicht. Das folgende Code-Snipplet zeigt, wie Einträge in die Liste hinzugefügt, geändert oder gelöscht werden können:

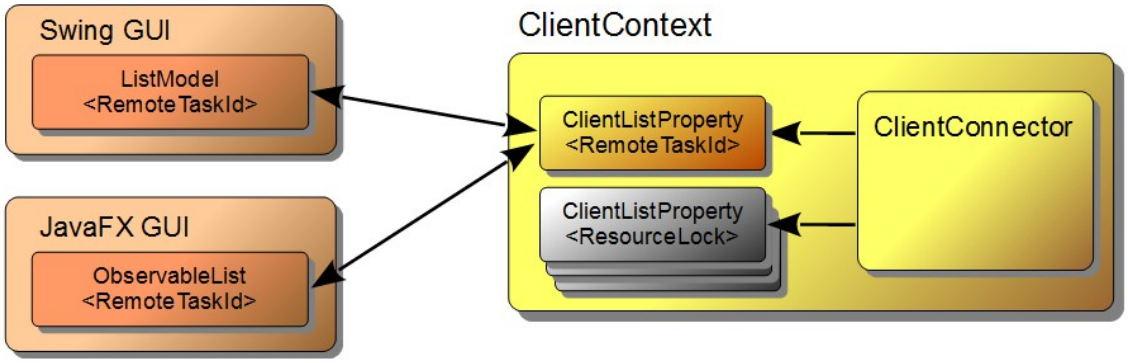
```
FXCheckListViewModel model = new FXCheckListViewModel(clientCtx);

model.init(); // fetches initial entries from server

ClientListProperty<String> listProp = model.remoteCheckList();
listProp.add("New task added"); // appends a new item
listProp.insertAt(0, "Inserted task"); // inserts a new item at index 0
listProp.replaceAt(1, "Replaced task"); // replaces item at index 1
listProp.removeAt(1); // removes item by index
listProp.remove("Inserted task"); // removes item
listProp.clear(); // removes all items
```

4.3 Remote-List-Properties im ClientContext

Im *ClientContext* kommen *ClientListProperty* Instanzen zum Einsatz, um aktuelle Zustände des Servers life abzubilden. Ein Beispiel dafür sind die laufenden *RemoteTasks* eines Clients, die im *ClientRemoteTaskManager* in einer *ClientListProperty<RemoteTaskId>* verwaltet werden. Diese Verwaltung ist GUI-neutral.

Wenn ein bestimmtes GUI-Framework die laufenden Tasks in einer GUI-Listenkomponente abbilden möchte, muss es dazu seine entsprechenden GUI-spezifischen List-Modelklassen befeuern. Dies sind bei **Swing** das *ListModel<T>*, bei **JavaFX** die *ObservableList<T>*.

Eine *ClientListProperty* erlaubt es mit ***setList(List)***, die verwaltete lokale Liste durch eine beliebige Andere zu ersetzen. So bepflanzt beispielsweise der **SwingClientContext** die *ClientListProperties* mit jeweils in einem *ListModelWrapper* eingehüllten *ListModel*, wohingegen der **FXClientContext** jeweils eine in einen *ObservableListWrapper* eingehüllte *ArrayList* verwendet. Auf diese Weise ist gewährleistet, dass jedes GUI-Framework auf jeweils seine Art mit Events befeuert wird.



„Bepflanzung" der ClientListProperties mit GUI-spezifischen List-Modellen.

5 Remote-Map-Properties

Ganz analog zu den Remote-*List*-Properties existieren Remote-*Map*-Properties zur zentralen Verwaltung von Key-Value Paaren. Hierfür gibt es folgende Implementierungen:

- ➔ **Serveseitig**
  ServerMapProperty<K,T>
- ➔ **Clientseitig**
  ClientMapProperty<K,T> (read-only)
  ClientModifiableMapProperty<K,T> (read-modify)

Remote-Map-Properties verwalten eine *Map<K,T>* Instanz und besitzen daher die typischen Map-

Manipulationsmethoden. Serverseitig sind dies *put(SessionId,K,T), remove(SessionId,K)* und *clear(SessionId)* und clientseitig entsprechend *put(K,T), remove(K)* und *clear()*.

Die Arbeitsweise einer *RemoteMapProperty* ist völlig analog zur *RemoteListProperty*. Daher wird hier auf eine weitere Beschreibung verzichtet und stattdessen auf die betreffende *Javadoc* Dokumentation verwiesen.

6 Besonderheiten

6.1 Besonderheiten einer ServerProperty

6.1.1 Aktivierung und Deaktivierung

Eine *ServerProperty* besitzt folgende Methoden, die ihren Lebenszyklus definieren:

- **init()**
  Bei diesem Aufruf wird die *ServerProperty* beim *ServerPropertyManager* registriert und steht fortan für Client-Zugriffe zur Verfügung. Die *init()* Methode wird typischerweise beim Start des Servers einmalig aufgerufen.
- **destroy()**
  Die *ServerProperty* deregistriert sich beim *ServerPropertyManager* und steht von nun an nicht mehr für Client-Zugriffe zur Verfügung. Der Wert der Property wird gelöscht und es werden keine weiteren Events mehr an Clients verschickt. Die *destroy()* Methode wird automatisch beim herunterfahren des Servers aufgerufen.

➔ *init()* kann nur einmal aufgerufen werden, *destroy()* hingegen beliebig oft.

6.1.2 Eingeschränkte Event-Benachrichtigung

Bei ihrer Initialisierung übernimmt eine *ClientProperty* den aktuellen Wert der zugehörigen *ServerProperty*. Bei dieser Gelegenheit wird die zugreifende *Session* bei der *ServerProperty* registriert. Letzterer ist also bekannt, welche Clients auf sie zugreifen.

Ändert sich nun der Wert der *ServerProperty*, dann sendet sie Aktualisierungs-Events nur an solche Sessions, die bei ihr registriert sind. Unbeteiligte Clients werden also nicht unnötigerweise mit Nachrichten behelligt, die sie gar nicht betreffen.

6.1.3 Permission-Kontrolle

Viele Anwendungen verlangen, dass bestimmte Benutzer nur bestimmte, für sie erlaubte Aktionen durchführen können. Der Esprit-Server beinhaltet für diesen Zweck ein umfangreiches Permission-Kontrollsystem, welches in dem Dokument „*EspritUserManagement*“ beschrieben ist. Der Zugriff auf Remote-Properties kann mit Hilfe dieses Permission-Systems kontrolliert werden.

Eine Server-Property kann mit oder ohne Zugriffskontrolle betrieben werden. Mit ***setPermissionControlled(true)*** wird die Permission-Kontolle eingeschaltet. Dann können nur noch solche Benutzer diese Property verwenden, denen vom Administrator explizit eine der folgenden Permissions vergeben wurde:

- **ServerPropertyPermission.READ**
  Erlaubt es dem Benutzer, den Wert einer Property zu lesen. Eine fehlende READ Permission führt dazu, dass ein Client immer den Wert *null* (bzw. leere Liste, leere Map) liest.

➔ **ServerPropertyPermission.MODIFY**
Erlaubt es dem Benutzer, den Wert einer Property zu ändern. Eine fehlende MODIFY Permission führt dazu, dass beim Versuch einer Änderung eine entsprechende *PermissionException* geworfen wird.

Eine *ServerProperty* mit eingeschalteter Permission-Kontrolle verlangt zwingend nach einem Benutzer-Login. Ohne dieses wird ein Änderungs-Request schon auf Clientseite mit einer einer entsprechenden Exception unterbunden.

➔ Für einen Administrator-Benutzer werden Permissions ignoriert; er besitzt per Definitionem stets alle Berechtigungen.

6.1.4 Server-Property mit Session-Objekten

Für die spezielle *SessionServerListProperty* gilt ein anderes Permission Konzept. Sie verwaltet Objekte vom Typ *SessionObject*, die einem bestimmten Benutzer „gehören“. Diese tragen die jeweilige *SessionId* des Benutzers. Beispiele sind die von einem Benutzer gesetzten *ResourceLocks* oder die von ihm gestarteten *RemoteTasks*. In einer *SessionServerListProperty* können viele Benutzer „ihre“ Objekte eintragen. Ein bestimmter Benutzer kann aber stets nur seine Eigenen sehen und manipulieren!

Eine *SessionServerListProperty* kann in zwei Modi betrieben werden, die mit ***setSessionControlMode(SessionControlMode)*** eingestellt wird:

➔ ***SessionControlMode.SESSION_BASED***
Nur die betreffende Client-Session wird über Änderungen informiert. Ein Benutzer-Login ist dabei nicht erforderlich.

➔ ***SessionControlMode.USER_BASED***
Hier ist ein Benutzer-Login erforderlich. Alle Clients mit dem gleichen Benutzer-Login werden über Änderungen informiert.

6.2 Besonderheiten einer ClientProperty

6.2.1 Aktivierung und Deaktivierung

Eine *ClientProperty* besitzt folgende Methoden, die ihren Lebenszyklus definieren:

➔ **goOnline()**
Bei diesem Aufruf holt die *ClientProperty* ihren initialen Wert vom Server und registriert sich als Listener für Werteänderungen. Von da an empfängt sie die entsprechenden Events und aktualisiert sich automatisch. Sie ist dann im *online* Zustand (*isOnline() == true*).

➔ **goOffline()**
Die *ClientProperty* deregistriert sich als Listener für Werteänderungen und erfährt keine weiteren Aktualisierungen. Sie ist dann im *offline* Zustand (*isOnline() == false*). In diesem Zustand ist ihr Wert nicht mehr synchron mit dem Wert der korrespondierenden *ServerProperty*.

➔ **destroy()**
Die *ClientProperty* geht offline und ihr Wert wird gelöscht. Sie kann nicht wieder aktiviert werden.

6.2.2 Automatische Aktivierung

Die Einstellung ***setActivationMode(ActivationMode)*** bestimmt, wann eine *ClientProperty* durch den Aufruf von *goOnline()* initialisiert wird. Es gibt die folgenden drei Möglichkeiten:

➔ *ActivationMode.**ON_CONNECT***
Die Property initialisiert automatisch beim Verbindungsaufbau. Dies ist nur sinnvoll bei Properties, die keine Authentifizierung erfordern. Sie geht offline nach dem Schließen der Verbindung.

➔ *ActivationMode.**ON_LOGIN*** (Standard-Einstellung)
Die Property initialisiert automatisch nach einem erfolgreichen Login. Sie geht offline nach einem Logout oder bei Verbindungsverlust.

➔ *ActivationMode.**NONE***
Die Property initialisiert nicht automatisch. Vielmehr muss ***goOnline()*** explizit zur Initialisierung aufgerufen werden. Sie geht offline durch den expliziten Aufruf von *goOffline()* oder bei Verbindungsverlust.

➔ Eine *ClientProperty*, die instantiiert wird, während eine Server-Verbindung bereits besteht, muss manuell durch den expliziten Aufruf von *goOnline()* initialisiert werden.

6.2.3 Wert einer offline-Property

Geht eine Property offline, sei es durch Verbindungsverlust, durch Logout oder den expliziten Aufruf von *goOffline()*, so wird sie ab diesem Zeitpunkt nicht mehr vom Server aktualisiert. Ihr Wert ist also nicht mehr synchron zum Wert auf dem Server. Daher wird er standardmäßig auf *null* gesetzt, bzw. geleert. Mit der Methode ***setAutoClearOnDisconnect(false)*** kann dies – falls gewünscht – verhindert werden, so dass der jeweils letzte Wert erhalten bleibt.

Umgekehrt kann eine Property durch die Einstellung ***setAutoDestroyOnDisconnect(true)*** bei Verbindungsverlust automatisch vernichtet werden. Sie kann dann nicht wieder aktiviert werden.

6.2.4 Direkte/indirekte Aktualisierung von Werten

Wird der Wert einer *ClientProperty* geändert, so wird **immer** zuerst serverseitig der Wert der zugehörigen *ServerProperty* geändert. Für die clientseitige Aktualisierung gibt es zwei verschiedene Techniken:

➔ **Lazy (reaktiv)**
Die lokale Änderung erfolgt als Reaktion auf das vom Server gesendete Update-Event.

➔ **Eager (direkt)**
Die lokale Änderung erfolgt unmittelbar auf die vom Server erhaltene Response. Das im Nachhinein vom Server gesendete Update-Event wird dann ignoriert. Auch bei langsamen Netzwerken sieht der verursachende Anwender auf diese Art dennoch eine schnelle Reaktionszeit.

Das Verhalten kann in der *ClientProperty* mit ***setUpdateEager(boolean)*** wie gewünscht eingestellt werden. Standardmäßig ist die **Lazy**-Aktualisierung eingestellt.

6.2.5 Sperren gegen andere Benutzer (mit *ResourceLock*)

Wenn ein Benutzer beispielsweise mit Hilfe einer *ClientListProperty* eine serverseitig zentral verwaltete Liste bearbeitet, dann möchte er ggf. während der Bearbeitung nicht durch die Aktivitäten anderer Clients gestört werden. Zu diesem Zweck kann er mit ***lockExclusive()*** eine exklusive Sperre (*ResourceLock*) beantragen, die ihm das ausschließliche Änderungsrecht garantiert. Mit ***lockShared()*** erhält er eine Sperre, die sicherstellt, dass die Property serverseitig nicht gelöscht wird. Der Aufruf ***lockRelease()*** beseitigt die Sperre wieder. Mit ***isLockedByMe()*** kann er fragen, ob er aktuell eine Sperre belegt hält.

Ein Client kann gleichzeitig nur eine Sperre belegen, entweder SHARED oder EXCLUSIVE. Sollte

er bereits eine SHARED Sperre halten, dann wird durch den Aufruf von ***lockExclusive()*** die vorhandene Sperre in eine vom Typ EXCLUSIVE umgewandelt – und umgekehrt.

➔ Ein Client sollte eine Sperre niemals länger als notwendig belegen, da dadurch die Arbeit von anderen Clients ggf. behindert wird.

ResourceLock-Sperren sind grundsätzlich „geleased". D. h. sie werden vom Server nach einer gewissen Zeit automatisch wieder aufgelöst, es sei denn, der Client re-triggert die Sperre rechtzeitig in regelmäßigen Zeitabständen. Dieses Retriggern geschieht automatisch, solange der Client online ist. Sollte der Client – aus welchen Gründen auch immer - offline gehen, dann wird der Server die Sperre nach dem ausgelaufenen Timeout auf jeden Fall auflösen.

6.2.6 Filterung der Listenwerte

Gelegentlich ist es erforderlich, dass eine *ClientListProperty* nur eine Submenge der Elemente einer *ServerListProperty* abbilden soll. Dazu kann ihr mit Hilfe der Methode ***setValueFilter(ListValueFilter)*** ein Filter gesetzt werden, welches die Werte nach beliebigen Kriterien filtert. Sie enthält dann nur noch solche Werte, die das Filterkriterium erfüllen.

➔ Fügt der Client Werte hinzu, die nicht dem Filterkriterium entsprechen, so werden diese in der korrespondierenden *ServerListProperty* zwar hinzugefügt, sind aber auf Clientseite nicht sichtbar!

6.2.7 Austausch der verwalteten Liste

Die *ClientListProperty* beinhaltet standardmäßig eine lokale *ArrayList*-Instanz, die automatisch mit der korrespondierenden *ServerListProperty* stets konsistent gehalten wird. Diese Instanz kann allerdings durch eine beliebige *List*-Implementierung ausgetauscht werden. Das folgende Beispiel zeigt, wie ein Swing-*ListModel* mit Hilfe einer *ClientListProperty* gemanaged werden kann. Dazu wird das *ListModel* in einen *ListModelWrapper* eingehüllt, der es als *List*-Implementierung erscheinen lässt. Der große Vorteil davon ist, dass eine Swing-*JList* Komponente, die dieses *ListModel* verwendet, bei jeder Änderung automatisch neu zeichnet.

```
DefaultListModel checkListModel = new DefaultListModel();
JList lockListView = new JList<>(checkListModel); // layouted GUI component

ClientListProperty propCheckList = new ClientListProperty<>(getClientContext(), "checkList");
propCheckList.setList(new ListModelWrapper<>(checkListModel)); // now the ListModel is managed
```

Ganz analog kann eine JavaFX-*ListView* Komponente angesteuert werden, wenn die *ClientListProperty* mit einer *ObservableList* Instanz versehen wird:

```
ObservableList<String> checkList = new ObservableListWrapper<>(new ArrayList<>());
ListView lockListView = new ListView<>(checkList); // layouted GUI component

ClientListProperty propCheckList = new ClientListProperty<>(getClientContext(), "checkList");
propCheckList.setList(checkList); // now the ObservableList is managed
```

Die *ClientListProperty* selbst ist unabhängig vom verwendeten GUI-Framework, kann auf diese Weise aber zur Ansteuerung verschiedener Frameworks verwendet werden. Dabei wird automatisch der Dispatching-Thread des jeweiligen Frameworks verwendet (*EventDispatching*-Thread bei *Swing*, *Platform*-Thread bei *JavaFX*).

7 Standard-Properties im Esprit-Server

Der Esprit-Server stellt eine Reihe von Standard-ServerProperties zur Verfügung, die für Clients direkt nutzbar sind. Es folgt eine Liste der Wichtigsten:

- **ESPRIT:GLOBAL_LIST** ServerListProperty<Object>
  Allgemein verfügbare zentral verwaltete Liste.
  Für den Zugriff sind Permissions erforderlich.
- **ESPRIT:GLOBAL_MAP** ServerMapProperty<String,Object>
  Allgemein verfügbare zentral verwaltete Map.
  Für den Zugriff sind Permissions erforderlich.
- **ESPRIT:ALL_TASKS** ServerListProperty<RemoteTaskId>
  Zeigt alle remoten Tasks, die auf dem Server laufen.
  Für den Zugriff sind Permissions erforderlich.
- **ESPRIT:USER_TASKS** ServerListProperty<RemoteTaskId>
  Zeigt die remoten Tasks benutzerbezogen.
  Ein bestimmter Benutzer kann nur seine eigenen Tasks sehen.
- **ESPRIT:ALL_LOCKS** ServerListProperty<ResourceLock>
  Zeigt alle Resource-Lock Sperren, die auf dem Server gesetzt sind.
  Für den Zugriff sind Permissions erforderlich.
- **ESPRIT:USER_LOCKS** ServerListProperty<ResourceLock>
  Verwaltet alle Resource-Lock Sperren benutzerbezogen.
  Ein bestimmter Benutzer kann nur seine eigenen Sperren sehen.
- **ESPRIT:ALL_WORKFLOWS** ServerListProperty<WorkflowStateModelWrapper>
  Zeigt alle remoten Workflows.
  Für den Zugriff sind Permissions erforderlich.
- **ESPRIT:USER_WORKFLOWS** ServerListProperty<WorkflowStateModelWrapper>
  Zeigt alle remoten Workflows benutzerbezogen.
  Ein bestimmter Benutzer kann nur seine eigenen Workflows sehen.
- **ESPRIT:COSERVER_CONNECTIONS** ServerListProperty<CoSessionId>
  Zeigt alle Verbindungen zu Co-Servern.
- **ESPRIT:COCLIENT_CONNECTIONS** ServerListProperty<CoSessionId>
  Zeigt alle Verbindungen zu Co-Clients.
- **ESPRIT:JAVA_VERSIONS** ServerListProperty<JavaVersion>
  Zeigt alle serverseitig installierten Java-Versionen.
- **ESPRIT:JAVA_VERSIONS_OF_COSERVERS** ServerMapProperty<String,List<JavaVersion>>
  Zeigt die auf den Co-Server installierten Java-Versionen.
- **ESPRIT:PROPERTY_INFOS** ServerMapProperty<String, ServerPropertyInfo>
  Zeigt Info-Objekte über alle zugreifbaren Server-Properties..

8 Vergleich mit anderen Frameworks

Im Markt gibt es mehrere Frameworks, die dazu dienen, Datenmodelle über Netz konsistent zu halten. Es gibt jedoch wesentliche Unterschiede in den Implementierungen, die gravierende Konsequenzen haben. Wir können im Wesentlichen zwei Konzepte unterscheiden:

- ➔ **Verteiltes Client-Model**
  Viele Model-Kopien jeweils auf den Clients
- ➔ **Zentrales Server-Model (mit Esprit-Server)**
  Model als Singleton-Instanz auf dem Server

8.1 Verteiltes Client-Model

Viele Clients besitzen die Kopie eines Models, das über Netz konsistent gehalten werden soll. Wenn ein Client seine Model-Kopie geändert hat, gibt er dies dem Server bekannt, der andere Clients **asynchron** aktualisiert.

Bei diesem Konzept fehlt die **synchrone** Komponente. Daher können grundsätzlich Konflikte auftreten, wenn zwei Clients gleichzeitig eine Änderung vornehmen. Dies kann z.B. bei der Verarbeitung von Listen zu Schwierigkeiten führen in der Form, dass Werte doppelt oder in falscher Reihenfolge eingefügt werden. Außerdem gibt es keine Möglichkeit für zentrale Prüfungen. Eine vom Client gemachte Änderung kann nicht vom Server verhindert werden, da sie bereits geschehen ist, bevor er davon erfährt.

8.2 Zentrales Server-Model (mit Esprit-Server)

Hierbei existiert nur eine einzige serverseitige Instanz eines Models. Alle Clients synchronisieren auf dessen Zustand. Zum Ändern des Models führt ein Client einen entsprechenden **synchronen Request** aus und erhält in der **Response** vom Server die Bestätigung, ob der Request erfolgreich war. Erst danach werden alle anderen Clients **asynchron** per **Event** benachrichtigt.

Dieses Konzept besitzt eine **synchrone** und eine **asynchrone** Komponente. Änderungen können auf dem Server nur synchron und nacheinander ausgeführt werden. Das Model ist dabei stets in einem eindeutigen Zustand, so dass Konflikte nicht entstehen können. Der verursachende Client erfährt bereits in der Response seine Bestätigung. Alle anderen Clients werden asynchron benachrichtigt. Da es eine zentrale Model-Instanz gibt, können Prüfungen zentral vom Server durchgeführt werden. Insbesondere sind auch komplexere Permission-Konzepte hier leicht einzubinden.

Darüber hinaus muss ein Client nicht notwendigerweise eine Kopie des Models besitzen, er kann sich vielmehr auf die Teil-Informationen beschränken, die ihn betreffen.