ESPRITSYSTEMS

Client/Server Netzwerk Programmierung mit Esprit

Übersicht über die Möglichkeiten der Client/Server Programmierung mit dem Esprit-Server

April 2017
Rainer Buesch

Inhaltsverzeichnis

1 Einführung

Dieses Dokument beschreibt die Funktionsweise des Esprit-Servers, sowie die Art der Client/Server Programmierung wie sie mit dem Esprit-Server möglich ist. Die folgenden Ausführungen erheben keinen Anspruch auf Vollständigkeit. Vielmehr geht es darum, dem Leser ein Grundverständnis für die Esprit-Technologie zu vermitteln und ihm eine Übersicht über deren Möglichkeiten zu geben.

1.1 Esprit Software

Durch seine besondere Architektur bietet der Esprit-Server Möglichkeiten, die mit herkömmlichen Middleware-Servern nicht oder nur sehr schwer realisierbar sind.

- ➔ Durch den integrierten Message-Service ist er in der Lage, Clients dynamisch über Änderungen aller Art (z.B. in der Datenbank) zu benachrichtigen. Alle Clients werden so – ohne eigenes Zutun – stets auf dem aktuellsten Stand gehalten. Jegliches sonst übliche Server-Polling entfällt.
- ➔ Die Kommunikation zwischen Client und Server geschieht über *die* **EMI**-Schnittstelle *(**E**sprit **M**ethod **I**nvocation)*. Diese Schnittstelle erlaubt es, Methoden auf dem Server aufzurufen, als seien es Lokale. Das Netzwerk ist für den Programmierer vollständig unsichtbar. Mit EMI können Client/Server Systeme extrem schnell, einfach und robust entwickelt werden, bei bestmöglicher Performance.
- ➔ Ein spezieller Transfer-Kanal erlaubt den Austausch großer Datenmengen zwischen Client und Server, ohne die sonstigen Funktionen zu stören. Über einen zusätzlichen Web-Kanal können auch HTML-Dokumente oder Bilddaten abgerufen werden. Nicht zuletzt dient er der zur dynamischen Aktualisierung der Client-Software.
- ➔ Besondere serverseitige ABO-Objekte (***A**live **B**usiness **O**bjects*) erlauben allen Clients eine einheitliche Sicht der Welt. Änderungen an solchen Objekten werden automatisch an alle Clients kommuniziert, die so - ohne selbst aktiv werden zu müssen - stets auf dem aktuellsten Stand sind.

Wegen dieser besonderen Merkmale ist der Esprit-Server ideal zur Unterstützung von Rich-Clients geeignet. Die Esprit Software beinhaltet den Esprit Middleware-Server, sowie ein umfangreiches Framework zur Entwicklung von Client/Server-Lösungen, sowohl für *Swing*- als auch für *JavaFX*-Benutzeroberflächen. Als „pure Java“ Software ist sie auf allen Plattformen lauffähig.

1.2 Funktionsprinzip

1.2.1 Zwei-Kanal-Verbindung

Eine wesentliche Besonderheit des Esprit-Servers besteht darin, dass Clients eine Doppel-verbindung über einen TCP Netzwerk-Port aufbauen:

- ➔ **Request/Response-Verbindung**
  Über diese Verbindung kann der Client Anfragen (Requests) absetzen und erhält eine entsprechende Antwort (Response). Die Abarbeitung des Requests geschieht synchron, d.h. der Client blockiert solange, bis er die Antwort erhalten hat.
- ➔ **Message-Verbindung**
  Über diese Verbindung empfängt der Client vom Server asynchrone Nachrichten (Messages). Damit ist der Server in der Lage, alle verbundenen Clients über Veränderungen (z.B. in der Datenbank) zu benachrichtigen. Inkonsistenzen werden so von vornherein vermieden.

1.2.2 Client-Sessions

Beide Netzwerkkanäle werden vom Server gemeinsam als jeweils eine Client-Session verwaltet. Da es sich um eine stehende TCP-Verbindung handelt, gibt es keine Verzögerungen durch vielfachen Verbindungs-Auf/Abbau. Ausgehende Client-Anfragen und eingehende Server-Nachrichten können problemlos zeitlich überlappen. *Requests/Responses* sowie *Messages* können jeweils beliebige Objekte als Argument transportieren. So geschieht der Datenaustausch zwischen Client und Server sehr einfach und performant.

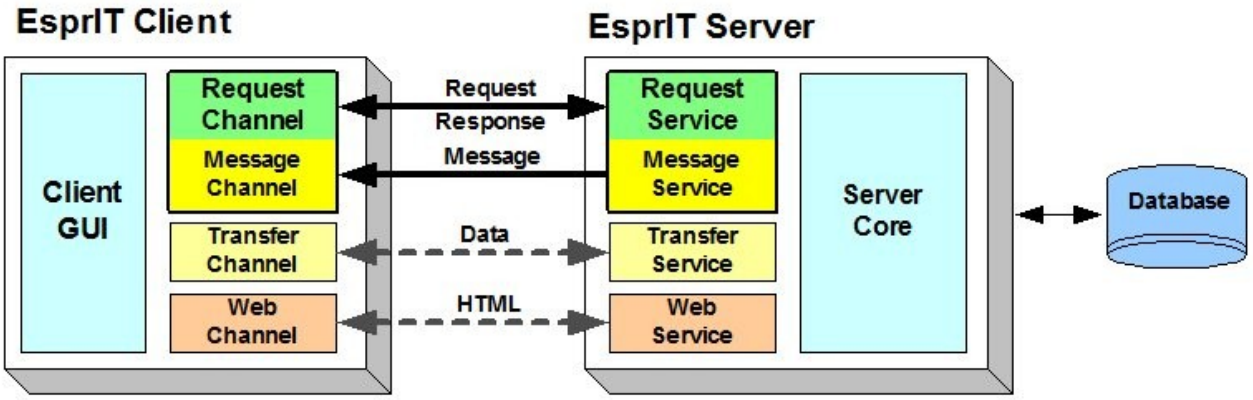


Bild 1) Die Verbindung des Clients zum Esprit-Server besteht aus einem Request/Response-Kanal, sowie einem Message-Kanal für asynchrone Server-Nachrichten. Ein Kanal für den Massendaten-Transfer sowie ein Webserver-Kanal werden bei Bedarf hinzugeschaltet.

1.2.3 Asynchrone Requests

Ein Request muss nicht notwendigerweise mit einer Response beantwortet werden, vielmehr kann die Antwort durch eine asynchrone Nachricht erfolgen, sobald sie fertig ist. Auf diese Weise funktionieren asynchrone Requests. Der Client muss dann nicht blockierend auf die Antwort warten, sondern wird informiert, wenn sie eintrifft.

1.3 Laufzeitumgebungen

Die clientseitige Laufzeitumgebung wird repräsentiert durch ein *ClientContext* Objekt, welches die Verbindung zu einem (oder mehreren) Esprit-Servern kapselt. Es ist das zentrale Objekt, über das der Programmierer Zugriff auf alle anderen Objekte der Clientseite hat. Die Instantiierung eines *ClientContext* Objekts entspricht dem Verbindungsaufbau zum Esprit-Server. Es können mehrere *ClientContext* Objekte in derselben VM (Virtual Machine) existieren.

Die serverseitige Laufzeitumgebung wird repräsentiert durch ein *ServerContext* Objekt, welches den Zugriff auf alle Serverkomponenten gestattet. Die Instantiierung des *ServerContext* Objekts entspricht dem Starten des Esprit-Servers.

2 EMI Schnittstelle

EMI (**E**sprit **M**ethod **I**nvocation) bildet die Schnittstelle zwischen Client und Server. EMI Aufrufe sehen ganz genauso aus, wie normale Java-Methodenaufrufe, werden allerdings vom *Esprit-Connector* jeweils über das Netzwerk an entsprechende API-Aufrufe im Server delegiert. Für den Programmierer ist das Netzwerk völlig unsichtbar. Der Server stellt nach Funktionalität gegliedert unterschiedliche APIs zur Verfügung.

2.1 Kommunikation mit EMI

Ein einfaches Beispiel für einen EMI-Aufruf soll die grundsätzliche Funktionsweise demonstrieren.

Betrachten wir einen Request, der vom Server das User Objekt für einen bestimmten Benutzer erfragt. Dabei wird ein vom Server bereitgestelltes sog. EPI-Interface[1] angefragt:

```
ClientContext ctx = getClientContext();
ServerUserEPI epi = ctx.getServerEPI(ServerUserEPI.class);
User user = epi.fetchUserByName("rainer");
```

Der *ClientContext* besitzt die Methode *getServerEPI(…)*, der ein Interface-Typ übergeben wird - in diesem Falle *ServerUserEPI.class*. Dieses Interface ist vom *UserManager* im Esprit-Server implementiert. Der Client generiert automatisch eine eigene Proxy-Implementierung des Interfaces, die alle dort definierten Methoden an die implementierende Instanz im Esprit-Server delegiert (in diesem Falle also an den *UserManager*). In Wirklichkeit findet also ein Aufruf über Netz (per Objekt-Serialisierung) an den Server statt. Das Netzwerk bleibt dem Programmierer dabei vollständig verborgen.

➔ Alle in einem EMI-Aufruf verwendeten Argumente und Rückgabewerte werden über Netz transportiert und müssen daher serialisierbar sein.

Die clientseitige Proxy-Implementierung (z.B. von *ServerUserEPI)* wird beim allerersten Aufruf automatisch erzeugt und wird zur schnellen Wiederverwendung in einem Cache gehalten.

2.1.1 EPI-Interfaces

Neben dem gezeigten *ServerUserEPI* Interface gibt es ca. 30 verschiedene Interfaces, die der Server bereitstellt. Zur besseren Identifizierung enden sie alle mit dem Suffix **-EPI** (**E**sprit **P**rogramming **I**nterface). Alle diese Interfaces definieren eine Reihe von Methoden, die in ihrer Summe die gesamte Funktionalität des Servers bereitstellen. Diese sind in der betreffenden Javadoc Dokumentation im Einzelnen beschrieben.

Der Server kann sehr einfach um weitere kundenspezifische Komponenten erweitert werden, die dann ihre eigenen EPI-Interfaces definieren.

2.1.2 Fehlerbehandlung

Die in EPI-Interfaces definierten Methoden-Deklarationen unterscheiden sich in keiner Weise von Standard-Java Methoden. Methoden können Exceptions werfen, müssen dies aber nicht! Eine Fehlerbehandlung geschieht auf exakt die gleiche Weise, wie bei der normalen Programmierung mit Java.

2.1.3 EMI Aufrufe mit Timeout

Bei einem EMI-Aufruf wartet der Client nicht beliebig lange auf die Antwort. Wenn der Server innerhalb von 15 Sekunden (Defaulteinstellung) nicht antwortet, bricht der Aufruf mit einer *TimeoutException* ab. Bei Bedarf kann der Timeout-Wert wie folgt beeinflusst werden.

```
ServerUserEPI epi = ctx.getServerEPI(ServerUserEPI.class, 2);  // timeout 2 sec
User user = epi.fetchUserByName("rainer");
```

In diesem Beispiel würde der Aufruf nach 2 Sekunden abgebrochen.

2.1.4 EMI-Aufrufe mit Priorität

EMI-Aufrufe können mit unterschiedlicher Priorität aufgerufen werden. Aufrufe mit hoher Priorität werden auf dem Server gegenüber den Requests anderer Benutzer bevorzugt abgearbeitet. Der

1 **EPI** steht für **E**sprit **P**rogramming **I**nterface. Im Grunde ist es nichts anderes als eine **API** (**A**pplication **P**rogramming **I**nterface), aber eben einer besonderen Art.

folgende Aufruf läuft mit erhöhter Priorität:

```
ServerUserEPI epi = ctx.getServerEPI(ServerUserEPI.class, RequestPriority.HIGH);
User user = epi.fetchUserByName("rainer");
```

Die möglichen Prioritäten sind HIGH, NORMAL und LOW, wobei NORMAL die Default-Einstellung ist. Sollen sowohl die Priorität, als auch der Timeout-Wert geändert werden, dann können einfach beide Werte als Argumente angegeben werden.

2.1.5 Rückgabewerte von EMI Aufrufen

Ein EMI Aufruf kann – wie jede Java Methode – primitive Werte oder Objekt-Referenzen zurückliefern. Sollten komplexere Strukturen zurückgeliefert werden, dann empfiehlt sich der Einsatz von speziellen Container-Objekten, die von der abstrakten Klasse ***EmiResponse*** abgeleitet sind, wie im folgenden Beispiel:

```
public final class AvailableServersResponse extends EmiResponse {
     private final List<ServerHost> serverList;
     public AvailableServersResponse(List<ServerHost> serverList) {
            this.serverList = serverList;
     }
     public List<ServerHost> getServerList() {
            return serverList;
     }
     public String doOnReceived() {
            logInfo("Number of available servers: "+serverList.size());
     }
}
```

Ein solches Objekt kann im Prinzip beliebige Daten zurückliefern. Das Besondere aber ist, dass beim Empfang automatisch die Methode ***doOnReceived()*** aufgerufen wird. Durch Überschreiben dieser Methode kann hier beliebige Logik eingebunden werden.

2.2 Asynchrone Requests mit EMI

Ein EMI-Request ist dann asynchron, wenn er durch eine asynchrone Message beantwortet wird. Der Aufrufer wartet also nicht, bis sein Aufruf vom Server beantwortet wurde, vielmehr erhält er eventuelle Ergebnisse in einer oder mehreren Messages zugeschickt, sobald diese fertig sind.

2.2.1 Beispiel für eine Asynchrone Anfrage

Im folgenden Beispiel wird per EMI-Aufruf ein *RoundTripJob* Objekt (abgeleitet von *EmiJob*) an den Server übergeben, welcher das Senden von 1000 Messages verursacht:

```
RoundTripJob rtJob = new RoundTripJob(ctx, 1000);
ctx.getServerEPI(ServerTestEPI.class).roundTrip(rtJob);
```

Als Folge dieses Aufrufs werden auf Serverseite 1000 Messages vom Typ *RoundTripMessage* asynchron erzeugt und gesendet. *RoundTripMessage* ist von der Mutterklasse ***EmiJobMessage*** abgeleitet und beinhaltet jeweils eine aufsteigende Sequenz-Nummer. Mit *isLast()* kann eine eintreffende Message befragt werden, ob sie die Letzte war.

➔ Man beachte, dass der gezeigte EMI-Aufruf nicht blockierend ist, da sowohl das Erzeugen, als auch das Senden der Messages auf Serverseite asynchron geschieht.

➔ *EmiJobMessages* werden grundsätzlich nur an den verursachenden Client geschickt!

2.2.2 Horchen auf EmiJobMessages

Man kann sich beim *EmiJob* als *Listener* registrieren, um die *EmiJobMessages* dieses Jobs zu empfangen und z.B. wie folgt zu reagieren:

```
RoundTripJob rtJob = new RoundTripJob(ctx, 1000);
rtJob.addEmiMessageListener(e → handleMessage(e));

ctx.getServerEPI(ServerTestEPI.class).roundTrip(rtJob); // execute request

private void handleMessage(EmiJobMessageEvent e) {
     if (e.isLast()) {
            System.out.println("Last message has seqNum: "+e.getSequenceNumber());
            System.out.println("JobId="+e.getJobId()+", duration="+e.getDuration());
     }
}
```

Jeder *EmiJob* hat eine eindeutige Nummer, die mit *getJobId()* erfragt werden kann. Die *EmiJobMessages* dieses Jobs besitzen jeweils die gleiche Nummer und sind damit diesem eindeutig zugeordnet. Mit *e.getDuration()* erfährt man, wie viele Millisekunden seit dem Absetzen des *EmiJobs* vergangen sind.

➔ Nach Eintreffen der letzten Message eines EmiJobs werden sämtliche dort registrierten Listener automatisch deregistriert.

2.2.3 Warten auf EmiJobMessages

Alternativ zum Horchen kann man sich auch auf *EmiJobMessages* synchronisieren, indem man einfach auf sie wartet, wie im folgenden Beispiel gezeigt:

```
RoundTripJob rtJob = new RoundTripJob(ctx, 1000);
ctx.getServerEPI(ServerTestEPI.class).roundTrip(rtJob);
...   // do something useful here
rtJob.awaitLastMessage(10, TimeUnit.SECONDS);
```

Der *EmiJob* selbst erfährt, wann die letzte Message eintrifft und befreit dann den aufrufenden Thread, der in *awaitLastMessage(...)* mit dem angegebenen Timeout wartet. Man beachte, dass der aufrufende Thread noch einiges „erledigen“ kann, **bevor** er sich in den Wartezustand begibt. Während dieser Zeit arbeiten Client und Server in Echtzeit **parallel**!

2.2.4 Gemischt synchrone und asynchrone EMI-Aufrufe

Ein EMI-Aufruf kann sowohl mit einer synchronen Response als auch mit einer oder mehreren asynchronen Messages beantwortet werden, wie dieses Beispiel zeigt:

```
RoundTripJob rtJob = new RoundTripJob(ctx, 1000);
RoundTripResponse response = ctx.getServerEPI(ServerTestEPI.class).roundTrip(rtJob);
System.out.println("Response duration: "+response.getDuration())
...   // process response data here
rtJob.awaitLastMessage(10, TimeUnit.SECONDS);
```

Das *RoundTripResponse* Objekt ist abgeleitet von *EmiJobResponse* und kann im Prinzip beliebige Ergebnisdaten des **synchronen Anteils** eines Requests beinhalten. Der **asynchrone Anteil** folgt danach durch das Verschicken der verlangten Anzahl von Messages. Im gezeigten Falle könnte der aufrufende Thread die Response-Ergebnisse bereits verarbeiten, bevor er anschließend auf die Message-Ergebnisse wartet.

2.3 Pre- und Postprocessing

2.3.1 Preprocessing von EmiJobs

Ein *EmiJob*, der als Argument eines EMI-Aufrufs mitgegeben wird, unterliegt einem Preprocessing, d. h. es wird automatisch in ihm die Methode ***doBeforeSend()*** aufgerufen. Durch Überschreiben dieser Methode kann dort beliebige Logik eingeklinkt werden. Während des Preprocessings stehen dabei mit *getClientContext()* alle Ressourcen des Contexts zur Verfügung.

2.3.2 Postprocessing von Response und Message

Sowohl die Klasse *EmiResponse*, als auch *EmiJobMessage* besitzen eine überschreibbare Methode ***doOnReceived()***, die automatisch aufgerufen wird, wenn die betreffende Instanz als Response bzw. als Message empfangen wurde. Die Default-Implementierung in *EmiJobMessage* sieht beispielsweise so aus:

```
protected void doOnReceived() throws Exception {
     if (isLast()) {
            logInfo("Duration of jobId " + getJobId()+": "+getDuration()+" ms");
     }
}
```

Hier kann also durch Überschreiben eine beliebige Logik eingeklinkt werden, so dass die Message praktisch selbst ihr mitgebrachtes Ergebnis auf Clientseite behandeln kann. Während des Postprocessings ist in der Message-Instanz der *ClientContext* gesetzt, so dass mit *getClientContext()* der Zugriff auf alle Ressourcen des Contexts möglich ist.

2.3.3 Temporäre Client Parameter

Angenommen, eine *EmiJobMessage* beinhaltet eine Meldung, die automatisch in einer Textkonsole der Applikation ausgegeben werden soll. Wie kann die eintreffende Message dann an die Textkonsole gelangen? In diesem Falle kommen sog. **Client-Parameter** zum Einsatz.

Client-Parameter werden von einem *EmiJob* in einem speziellen Cache des Clients abgelegt und sind solange gültig, bis die letzte Message der asynchronen Antwort angekommen ist. Das folgende Beispiel zeigt das Ablegen einer Textkonsolen-Referenz im Konstruktor des *PrintJob*'s:

```
public PrintJob(ClientContext ctx, TextConsole console) extends EmiJob {
     super(ctx) {
            putClientParam("TEXT_CONSOLE", console);
     }
}
```

Eine vom *PrintJob* veranlasste *PrintMessage* könnte dann wie folgt auf die Textkonsole zugreifen:

```
public PrintMessage(...) extends EmiJobMessage {
…
     @CalledAsync
     protected void doOnReceived() {
            TextConsole console = getClientParam("TEXT_CONSOLE");

            Runnable r = () → console.println("Here is my result");
            getClientContext().getGuiDriver().invokeLater(r);
     }
}
```

➔ Zu beachten ist, dass die *doOnReceived()* Methode in einer Message asynchron aufgerufen wird. Sollten dort Aktionen erfolgen, die GUI-Komponenten betreffen, so müssen diese - wie gezeigt - mit

invokeLater() an den EventDispatcher-Thread übergeben werden.

➔ Die letzte aus einem *EmiJob* resultierende Message löscht sämtliche gespeicherten Client-Parameter dieses Aufrufs.

2.4 Vergleich zu RMI

Wer mit dem Java Standard **RMI** (**R**emote **M**ethod **I**nvocation) vertraut ist, wird aufgefallen sein, dass **EMI** dem ziemlich ähnlich ist. Es gibt allerdings einige erhebliche und wichtige Unterschiede:

- ➔ EMI benötigt keinen Namingservice (z.B. RMIRegistry).
  Die clientseitigen Proxy-Implementierungen müssen also nicht erst heruntergeladen werden, sondern werden dynamisch im Client erzeugt. Dies ist erheblich performanter.
- ➔ EMI-Methoden müssen nicht mit Exceptions deklariert werden.
  RMI Schnittstellen verlangen stets zwingend den Umgang mit checked Exceptions.
- ➔ EMI-Aufrufe funktionieren über die Esprit-Netzwerkverbindung und sind damit **internetfähig**. Bei RMI dagegen besitzt jedes Server-Objekt eine andere (unbekannte) Portnummer. Da Firewalls typischerweise diese Portnummern nicht freischalten, ist RMI nicht internetfähig. Man hilft sich mit Webclients, die aber in Funktionalität mit Rich-Clients nur schwerlich konkurrieren können.
- ➔ EMI-Aufrufe können auch asynchron ausgeführt werden (Response per Message)
  RMI bietet dazu keine vergleichbare Möglichkeit.
- ➔ EMI-Aufrufe benutzen eine stehende Verbindung und bieten daher höchste Performance.
  Bei RMI wird hingegen für jeden Methodenaufruf eine eigene Verbindung auf- und wieder abgebaut. Dies führt zu Performanceproblemen bei häufigen Aufrufen.

3 Model-View-Controller Prinzip über Netz

Das **M**odel-**V**iew-**C**ontroller Prinzip (MVC) wird üblicherweise in GUIs angewendet, um z.B. die Zustände von GUI-Komponenten zu kontrollieren. Weil es äußerst robust funktioniert, wird es sehr gerne verwendet, war bisher aber auf lokale Anwendungen beschränkt. Da der Esprit-Server einen integrierten Message-Kanal besitzt, ist es mit ihm erstmals möglich, dieses Prinzip auch über Netz zu realisieren. Die sich daraus ergebenden Möglichkeiten sind enorm.

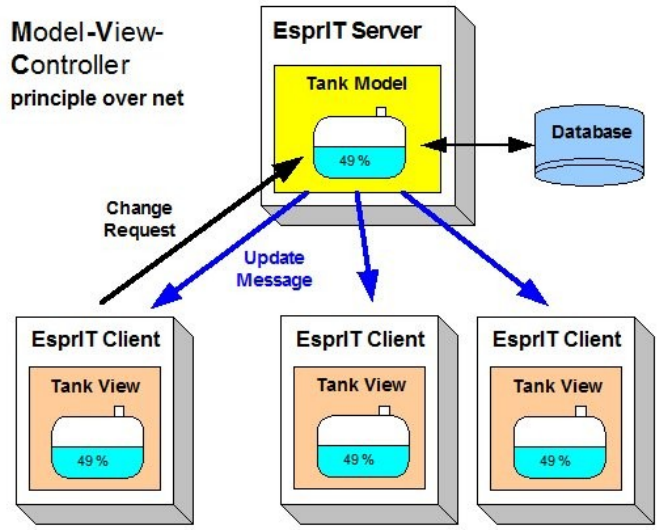


Bild 3) Das Model-View-Controller-Prinzip kann nun über Netz realisiert werden

3.1 Server-Modelle

ServerModel Instanzen sind zentrale Objekte, die auf Serverseite (typischerweise als Singleton Objekte) existieren. Sie bilden einen Zustand ab, der von Clients life beobachtbar ist. Im folgenden wird ein einfaches Beispiel betrachtet, aber der Komplexität sind hier kaum Grenzen gesetzt. Lassen Sie uns das MVC-Prinzip anhand des *Position-Tools*, welches im Esprit-Server als Beispiel enthalten ist, gedanklich durchspielen.

3.1.1 PositionServerModel

Grundlage des Prinzips ist das *PositionServerModel*, abgeleitet von *AbstractServerModel*, welches in unserem Beispiel eine Position als veränderlichen x/y Wert, speichert. Mit der Methode *setPosition(x,y)* kann dieser Wert verändert und mit *getPosition()* erfragt werden. Der Esprit-Server hält eine Instanz des *PositionServerModels* im *ServerContext* vor. Der Java Code des *PositionServerModels* sieht folgendermaßen aus:

```
public final class PositionServerModel extends AbstractServerModel<ServerContext>
                                       implements PositionEPI {
    private Point currentPos = new Point(0, 0);

    @Override
    public void init() throws Exception {
          super.init();
          registerServerEPI(PositionEPI.class, this);
    }

    @Override
    public synchronized Point getPosition() {
          return currentPos;
    }

    @Override
    public synchronized void setPosition(Point point) {
          currentPos = point;
          sendChangeEvent(new PositionEvent(getAccessor(), currentPos));
    }
}
```

Das Modell hat als einzige Zustandsvariable *currentPos* ein *Point*-Objekt. Dieses kann mit *getPosition()* erfragt und mit *setPosition(Point)* geändert werden. Wie aus dem Quellcode ersichtlich wird bei jeder Veränderung der Position ein *PositionEvent* an alle Clients gesendet. Dieses beinhaltet die neue Position, sowie die *SessionId* des zugreifenden Clients, der die Aktion ausgeführt hat. Letztere ist dem Server stets bekannt und kann mit *getAccessor()* erfragt werden.

➔ Man beachte, dass die beiden Methoden *getPosition()* und *setPosition(x,y)* als *synchronized* deklariert sind. Dies ist wichtig, da im Allgemeinen viele Clients gleichzeitig versuchen könnten, die Position zu verändern. Da jede Client-Aktion im Server in einem anderen Thread läuft, muss der Zugriff synchonisiert erfolgen.

3.1.2 Das PositionEvent

Der Quellcode des *PositionEvents* sieht folgendermaßen aus:

```
4 public class PositionEvent extends AbstractSessionEvent {
    private final Point point;
5   public PositionEvent(SessionId sessId, Point point) {
          super(sessId);
          this.point = point;
    }
6   public Point getPosition() {
          return point;
    }
7   public interface Listener extends GenericEvent.Listener {
          public void positionChanged(PositionEvent e);
    }
  }
```

Das Event überträgt den aktuellen Zustand (die Position) des *ServerModells* an die Clients. Durch das innere *Listener*-Interface ist vorgeschrieben, mit welcher Methode ein Client (die View-Klasse) auf dieses Event zu reagieren hat.

7.1.1 Implementierung des EPI-Interface

Entscheidend wichtig ist das implementierte EPI-Interface, welches die für die Clients sichtbaren Methoden definiert. Es sieht wie folgt aus:

```
public interface PositionEPI extends ServerEPI {
    void setPosition(Point point);
    Point getPosition();
}
```

Dieses Interface wird in der *init()* Methode des *PostionServerModels* zusammen mit der implementierenden Instanz selbst im Server registriert und ist dann fortan von Clients abrufbar.

7.1.2 Client Zugriffe

Auf der Clientseite sieht der Zugriff zur Abfrage der Position dann wie folgt aus:

```
ClientContext ctx = getClientContext();
Point position = ctx.getServerEPI(PositionEPI.class).getPosition();
```

Entsprechend zum Setzen einer neuen Position:

```
ctx.getServerEPI(PositionEPI.class).setPosition(new Point(10,15));
```

7.2 Clientseitiger View

Jedem GUI-Programmierer ist vertraut, wie man z.B. eine Schaltfläche mit einer auszuführenden Callback-Funktion koppelt. Dies funktioniert ganz ähnlich mit den über Netz empfangenen Events des Esprit-Servers. Um die jeweils aktuelle Position im Client-GUI anzuzeigen konstruieren wir das *PositionLabel*, eine einfache *JLabel*-Komponente, die auf *PositionEvents* reagiert und die aktuelle Position als Text anzeigt:

```
public class PositionLabel extends JLabel {
    private final PositionEvent.Listener listener = e → positionChanged(e);
    public PositionLabel(ClientContext ctx) {
          ctx.getClientConnector().addNetEventListener(listener);
    }
```

```
    private void positionChanged(PositionEvent e) {
        setPositionText(e.getPosition());
    }
    private void setPositionText(Position p) {
        setText(p.getX()+","+p.getY());
    }
}
```

Wie Sie sehen, registriert sich die Komponente beim *ClientConnector*, um auf Events zu horchen. Dort werden alle erdenklichen Event-Typen vom Server empfangen, da wir aber explizit *PositionEvent.Listener* implementieren, horchen wir gezielt auf *PositionEvents*. Beim Empfang eines solchen Events wird die dafür vorgeschriebene Methode *positionChanged(e)* aufgerufen, in der formuliert ist, wie die Reaktion zu erfolgen hat: hier das einfache Setzen des Textes.

➔ Es ist guter Programmierstil (seit Java-8), den Listener als Lambda-Expression zu implementieren.

Unser *PositionLabel* hat noch einen kleinen Fehler: der richtige Wert wird erst dann angezeigt, wenn das erste *PositionEvent* empfangen wurde. Es fehlt noch die anfängliche Initialisierung auf den aktuellen Wert. Dazu muss die aktuelle Position des *PositionServerModels* einmal erfragt werden. Seine Implementierung ist trivial. Die Initialisierung geschieht typischerweise im Konstruktor und könnte dann so aussehen:

```
    public PositionLabel(ClientContext ctx) {
        ctx.getClientConnector().addNetEventListener(listener);
        setPositionText(ctx.getServerEPI(PositionEPI.class).getPosition());
    }
    ...
```

7.3 Große Server-Modelle

Im obigen Beispiel wurde ein sehr einfaches *ServerModel* mit einem einzigen Parameter gezeigt. In der Praxis kann ein solches Modell beliebig komplex werden. So könnte man beispielsweise eine komplette Fabrik mit allen ihren Fertigungsarbeitsplätzen in einem *FactoryServerModel* beschreiben. Dieses Modell hätte viele Parameter und würde entsprechend viele verschiedene Event-Typen senden. Clients könnten das Modell teilweise oder ganz visualisieren, je nachdem für welche Events sie sich interessieren. Entscheidend ist, dass alle Clients stets den jeweils aktuellen Zustand darstellen – und dies mit bester Performance, da sie vom Server immer nur die notwendigste Delta-Information empfangen.

7.4 Alive Business Objects

Eine besonders interessante Anwendung des MVC-Prinzips sind die „lebenden“ Business-Objekte (***A**live **B**usiness **O**bjects* = ABOs). Im Unterschied zu Server-Modellen kann es davon mehrere Instanzen desselben Typs geben, die sich dann aber eindeutig durch einen Primary-Key unterscheiden. Ein einmal erzeugtes ABO wird in einem speziellen Server-Cache gespeichert und bleibt dort existent solange ein Client-Zugriff darauf hat.

7.4.1 Ein ABO Beispiel

Nehmen wir als Beispiel ein *Tank*-Objekt an, welches aus der Datenbank gelesen wurde, also persistent ist (abgleitet von *AbstractPersistentABO*). Nehmen wir ferner an, dass ein Client (Controller) den Füllstand des Tanks ändert, indem er per Request in diesem Objekt die *setLevel(value)* Methode aufruft. Ähnlich wie ein *ServerModel* reagiert das *Tank*-Objekt, indem es den neuen Wert in der Datenbank speichert und dann eine Nachricht an alle Clients verschickt, in

der die Änderung kommuniziert wird.

➔ Alle Clients erhalten die Änderung sofort, ohne selbst eine neuerliche Datenbankabfrage machen zu müssen!

Man beachte ferner, dass ein anderer Client, der z.B. die Art der Flüssigkeit verändern möchte, das *Tank*-Objekt nicht nochmals aus der Datenbank lesen muss, sondern es bereits im serverseitigen Cache vorfindet. Er kann auch mehrere Eigenschaften des Objekts gleichzeitig ändern, bevor er es wieder in die Datenbank zurückschreibt. In der Summe ergeben sich durch diesen Mechanismus enorme Performance-Vorteile gegenüber herkömmlichen Techniken[2].

7.4.2 Client-Zugriff auf ABOs

Der Zugriff auf solche ABOs gestaltet sich für den Programmierer sehr einfach. Folgendes Beispiel zeigt, wie das *Tank* Objekt ausgehend von einer Master-Instanz des betreffenden ABOs vom Server geholt und verändert wird:

```
TankABO master = TankABO.getMaster(clientCtx);
TankABO t = master.fetch(false, 100);  // fetch instance from server by primary key
t.setLevel(1500);
t.setFluid("Water");
t.update();       // update on server
t.drop();         // switch offline - update messages are not received any more
```

Zunächst wird eine Master-Instanz vom Typ *Tank* erzeugt, die in der Lage ist, das „echte“ Business-Objekt vom Server zu holen. Dazu wird als Argument von *fetch(false, pk)* der Primary-Key angegeben (hier die Tanknummer 100), der auch aus mehreren Werten bestehen kann. Anhand dieses Schlüssels wird die Instanz im ABO-Cache des Servers gesucht. Falls sie bereits existiert, erhält der Client sie sofort. Ansonsten wird sie frisch aus der Datenbank gelesen und instantiiert.

Würde *fetch(...)* mit dem Argument *true* aufgerufen, dann würde das ABO gleich mit einer Resource-Sperre belegt, um zu verhindern, dass andere Benutzer es gleichzeitig verändern können (siehe: Resource-Sperren). Andere Clients haben dann nur lesenden Zugriff.

Das Beispiel zeigt auch die Änderung von gleich mehreren Eigenschaften, die dann mit *update()* auf den Server zurückgeschrieben werden. Daraufhin – und das ist das Neue an dieser Technik – werden alle Clients, per Nachricht informiert und erhalten so umgehend die geänderten Werte, ohne eigenes Zutun. Der abschließende *drop()* Aufruf führt dazu, dass der Client das Business-Objekt offline schaltet. Es wird dann nicht mehr vom Server aktualisiert. Auf ähnlich einfache Weise geschieht auch das Erzeugen neuer, bzw. das Löschen vorhandener *Tank*-Objekte.

8 Server Kaskadierung

8.1 Zusammenschalten von Servern

Ein Server kann sich mit einem oder mehreren anderen Servern, seinen sog. *Co-Servern*, verbinden und ist dann deren *Co-Client*. Auf diese Weise sind Server kaskadierbar und können zu komplexen Verbundnetzen zusammengeschaltet werden, wobei jeder Einzelne typischerweise auf einem anderen Rechner läuft und auch andere Aufgaben haben mag.

---

2 Dies funktioniert völlig anders bei handelsüblichen Applikations-Servern, die auf RMI (Remote Method Invocation) basieren. Dort muss vor jedem Zugriff das Objekt aus der Datenbank gelesen und nach dem Zugriff wieder in die Datenbank zurückgeschrieben werden, und dies für die Änderung jeder einzelnen Objekt-Eigenschaft – ein altbekanntes, aber konzeptionsbedingtes Performanceproblem.

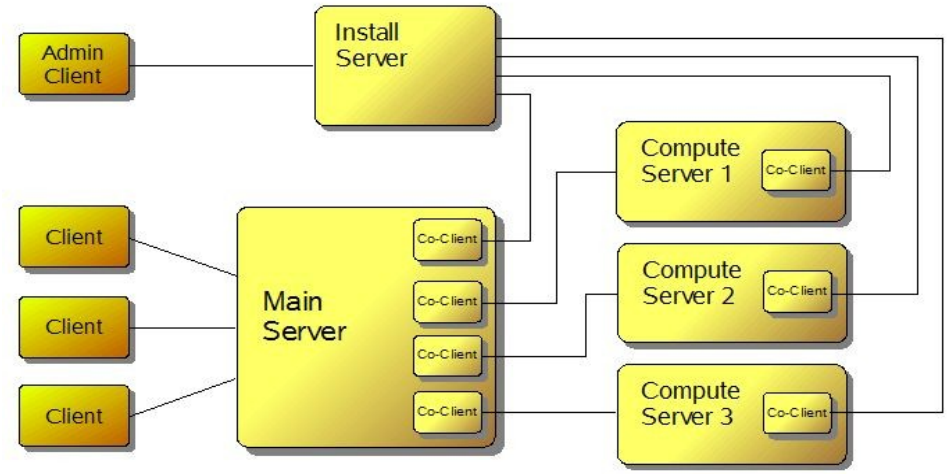


Bild 4) Mehrere Server lassen sich zu einem Verbundnetz kaskadieren. Client-Anfragen können entweder direkt an einen bestimmten Ziel-Server „geroutet“ oder von Server zu Server durchgereicht werden.

Per Konfiguration ist jedem Server bekannt, wer seine *Co-Server* und *Co-Clients* sind, so dass er sich nach zeitweiligem Netzwerkausfall oder nach einem Restart automatisch wieder korrekt in das Netzwerk einklinkt. Die Verbindung zwischen *Co-Client* und *Co-Server* ist in sofern eine Besondere, als sie ohne eine Benutzerauthentifizierung besteht und ihr damit keine Permission-Schranken auferlegt sind.

8.2 Weiterreichen von Requests

In einem Server-Verbund können Requests vom Client entweder direkt an einen Zielserver geschickt, oder aber von Server zu Server weitergereicht werden, wobei jeder Server seinen speziellen Teil zur Antwort (Response) beiträgt.

Beispielsweise kann der Zentralserver die Datenbankabfrage erledigen (denn er hat die Datenbankverbindung) und dann das mit Zwischenergebnissen gefüllte Response-Objekt weitersenden an einen Compute-Server, der in der Lage ist, spezielle geometrische Berechnungen durchzuführen. Wenn der Client seine Response schließlich zurückerhält, haben mehrere Server für ihn daran gearbeitet.

Einer der Vorteile dieses Konzeptes liegt darin, dass Aufgaben auf verschiedene Server verteilt werden können. Dabei brauchen die involvierten Server keine spezielle Intelligenz zu besitzen. Für die Praxis bedeutet dies, **dass alle Server – bis auf Unterschiede in ihrer Konfiguration – identisch sind**.

Eine weitere interessante Möglichkeit besteht darin, dass Dateien quer über den Server-Netzverbund quasi „geschoben“ werden können (siehe: *FileTransfer-Mechanismus*). Sollte beispielsweise ein Compute-Server ein Berechnungsergebnis fertiggestellt haben, deponiert er es in Form von Ergebnisdateien auf dem Zentral-Server, der seinerseits die Clients veranlasst, Letztere dort abzuholen.

Ein solcher Server-Verbund kann auch eingesetzt werden, um z.B. diverse Prozesse auf den jeweiligen Plattformen zu überwachen und deren Zustände an interessierte Clients durch zu melden. Die Anwendungsmöglichkeiten sind sehr vielfältig.

9 Architektur des Esprit-Servers

Dieses Kapitel bietet einen Überblick über die wichtigsten Funktionen des Esprit-Servers. Das in der Software enthaltene *Admin-Tool*, ein spezieller Administrations-Client, spiegelt diese wider und

ermöglicht dem Administrator die visuelle Kontrolle über die meisten Vorgänge im Server.

9.1 Skalierbarkeit

Ein Server muss viele Clients gleichzeitig bedienen. Er tut dies, indem er die einzelnen Requests in Threads gleichzeitig laufen lässt. Sollten dabei Serverobjekte manipuliert werden, sorgt er für eine saubere Synchronisierung der Zugriffe.

Der Esprit-Server ist hochgradig „multithreaded“, d.h. in ihm laufen viele Threads gleichzeitig. Zur Synchronisierung verwendet er die modernen Concurrency-Klassen (seit Java-6), mit deren Hilfe dies sehr robust und performant möglich ist.

Threads sind „teure“ Ressourcen. Der Server achtet darauf, möglichst keine neuen Threads zu erzeugen, sondern stattdessen bereits vorhandene wiederzuverwenden. Dazu verwaltet er sie in sogenannten *Thread-Pools* und kann so sehr genau steuern, wie viele Threads wirklich gleichzeitig laufen, um sich letztendlich vor Überlastung zu schützen.

Kurz gesagt: Der Esprit-Server ist **skalierbar**, d.h. der Administrator kann über die Konfiguration der Thread-Pools den Server optimal an die Hardware, auf der er läuft, anpassen. Einem PC kann man natürlich nicht soviel Last zumuten, wie einer Multi-CPU-Maschine, aber – und das ist entscheidend – der PC ist jederzeit austauschbar gegen einen Hochleistungsrechner, die Software aber bleibt exakt die gleiche.

Eine optimale Einstellung der Thread-Pools zu finden erfordert Experimentierfreude und etwas Erfahrung. Je nach Anforderung der laufenden Applikationen kann sich das Optimum aber auch zeitlich verschieben. Deshalb sind die entscheidenden Parameter zur Laufzeit dynamisch veränderbar.

9.2 Erweiterbarkeit

Sämtliche Server-Funktionalität ist gekapselt in der eingangs erwähnten *ServerContext* Klasse. Sie bietet Zugriff auf alle enthaltenen Server-Komponenten. Die Instantiierung dieser Klasse entspricht dem Hochfahren des Servers.

Ggf. ist es notwendig den Esprit-Server um kundenspezifische Funktionalitäten zu erweitern, um beispielsweise ein spezielles Permission-System einzubauen. Dies kann sehr einfach geschehen durch das Erstellen einer spezialisierten *MyServerContext* Klasse, die sich von *ServerContext* ableitet. Zudem können sie bestimmte Server-Komponenten durch ihre Eigenen ersetzen, der *Esprit*-Server bietet dafür eine ganze Reihe von Schnittstellen. Statt des standardmäßigen *UserManagers* verwenden Sie z.B. ihren *MyUserManger,* abgeleitet von *UserManager,* der die Benutzer in Ihrer Datenbank verwaltet.

9.3 Logfile Management

9.3.1 Log-Dateien

Der *Esprit*-Server verwaltet eine Reihe von Logdateien in denen sämtliche Serveraktivitäten vollständig nachvollziehbar sind. Ein Client kann den Inhalt einer Logdatei zur Ansicht anfragen. Damit dies mit guter Performance geschieht, ist die Größe einer Logdatei begrenzt. Beim Erreichen der maximalen Größe schließt der Server die aktuelle Logdatei und legt automatisch eine Neue an. Auch der Detaillierungsgrad der Logmeldungen ist einstellbar. Für den Testbetrieb wählt man den gesprächigen *verbose*-Mode, im Produktionsbetrieb genügt es, wenn Aktivitäten im info-Mode protokolliert werden. Die möglichen Modi entsprechen den zur Verfügung stehenden *LogLevels* eines *LogChannels*.

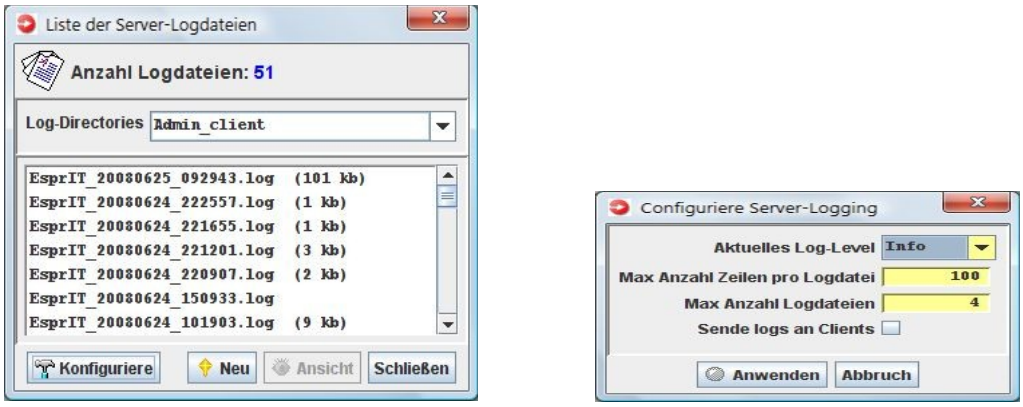


Bild 5) Der Log-Konfigurationsdialog ermöglicht direkte Einsicht in die Logdateien sowie das dynamische Verändern der Logging-Parameter.

9.3.2 Log-Kanäle

Würden alle Log-Einträge in die gleiche Datei erfolgen, dann wäre sehr schwer nachvollziehbar, welcher Client welche Serveraktivität verursacht hat. Deshalb unterscheidet der Server die Clients nach deren *Application-Name* und öffnet jeweils einen Client-spezifischen Log-Kanal, in dem nur die Aktivitäten eines bestimmten Client-Typs protokolliert werden.

9.3.3 Automatische Log Bereinigung

Damit das Log-Filesystem nicht vollläuft, ist die Anzahl der Logdateien begrenzt. Ein Hintergrundprozess löscht automatisch in regelmäßigen Zeitabständen die jeweils ältesten Logdateien, und zwar in allen geöffneten Log-Kanälen. Sämtliche Veränderungen an Log-Dateien oder Kanälen sind von dafür registrierten Clients beobachtbar, da sie per Nachricht mitgeteilt werden. Auf Wunsch (z.B. im Admin-Client) werden sogar die einzelnen Logmeldungen versendet und sind dann in einer *Server-Logkonsole* auf Clientseite unmittelbar zu verfolgen.

9.4 Zeitgesteuerte Server-Prozesse

Der *Esprit*-Server bietet die Möglichkeit, Kundenspezifische Klassen (abgeleitet von *AbstractServerBatchTask*) in einer Prozess-Liste des Servers zu registrieren, jeweils mit einer zusätzlichen Angabe, wann und mit welcher Wiederholungsrate sie aufgerufen werden sollen. Ein spezieller Thread des Servers führt sie dann zum gegebenen Zeitpunkt aus. Es kann jeweils nur ein zeitgesteuerter *ServerBatchTask* gleichzeitig laufen. Weitere Prozesse, die auch laufen möchten, blockieren in einer Warteschlange bis der Ausführungs-Thread wieder frei ist.

Ein laufender *ServerBatchTask* sendet dem Client regelmäßige Fortschrittsmeldungen, so dass dieser den Ablauf in einem Laufbalken visualisieren kann. Auch von den Logmeldungen, die der Server-Prozess in die Logdatei schreibt, erhält der Client eine Kopie per Nachricht, und kann sie deshalb in einer Konsole anzeigen. Es steht dem Client jederzeit frei, einen laufenden *ServerBatchTask* abzubrechen, manuell neu zu starten oder ihn dynamisch umzukonfigurieren. In einer speziellen Logdatei wird mitgeschrieben, wann welcher *ServerBatchTask*, gelaufen ist, und mit welchem Ergebnis er abgeschlossen hat.

Bild 6) Mit diesem Dialog kann ein ServerBatchTask konfiguriert und manuell ausgeführt werden.

Eine typische Anwendung besteht z.B. darin, dass regelmäßig bestimmte Datenübertragungen (z.B. Datenabgleich von einer Datenbank mit einer anderen) ausgeführt werden.

9.5 Datenbank-Verbindungen

Per Konfiguration kann dem Esprit-Server mitgeteilt werden, dass er sich mit einer oder mehreren relationalen Datenbanken verbinden soll. Diese können sogar von verschiedenen Herstellern stammen, denn die zugrunde liegende Persistenzschicht (*DBObjects*) basiert auf JDBC und ist damit weitestgehend datenbankunabhängig.

Die Verbindung zu einer Datenbank erfolgt über einen konfigurierbaren *Connection-Pool*, der die gleichzeitige Ausführung mehrerer Transaktionen von verschiedenen Benutzern gestattet. Da Verbindungen zu mehreren Datenbanken gleichzeitig bestehen können (also mehrere Connection-Pools!), ist es möglich, Datensätze aus einer Datenbank zu lesen, um sie unmittelbar in eine andere einzufügen.

➔ Damit hat man praktisch eine on-the-fly Datenübertragung zwischen zwei ggf. sehr unterschiedlichen Datenbanksystemen realisiert, die zudem außerordentlich performant ist.

Der *Esprit*-Server selbst ist nicht auf eine Datenbank angewiesen.

9.6 Server Aktivität

9.6.1 Server-Statistik

Der Esprit-Server führt interne Statistiken über aller erdenklichen Ereignisse. Ein spezielles *ServerStatisticABO* liefert dem Client in regelmäßigen Zeitabständen Schnappschüsse des Momentanzustands und macht so die Servertätigkeit für den Client beobachtbar. Auf diese Weise findet man leicht Engpässe, die ggf. eine Umkonfiguration ratsam erscheinen lassen. Wenn beispielsweise entdeckt wird, dass die Ausführung von Requests öfter abgelehnt wurde, dann könnte man die maximale Anzahl parallellaufender Requests erhöhen. Um gezielte Messungen durchzuführen, lassen sich alle Zählerwerte der Statistik zurücksetzen.

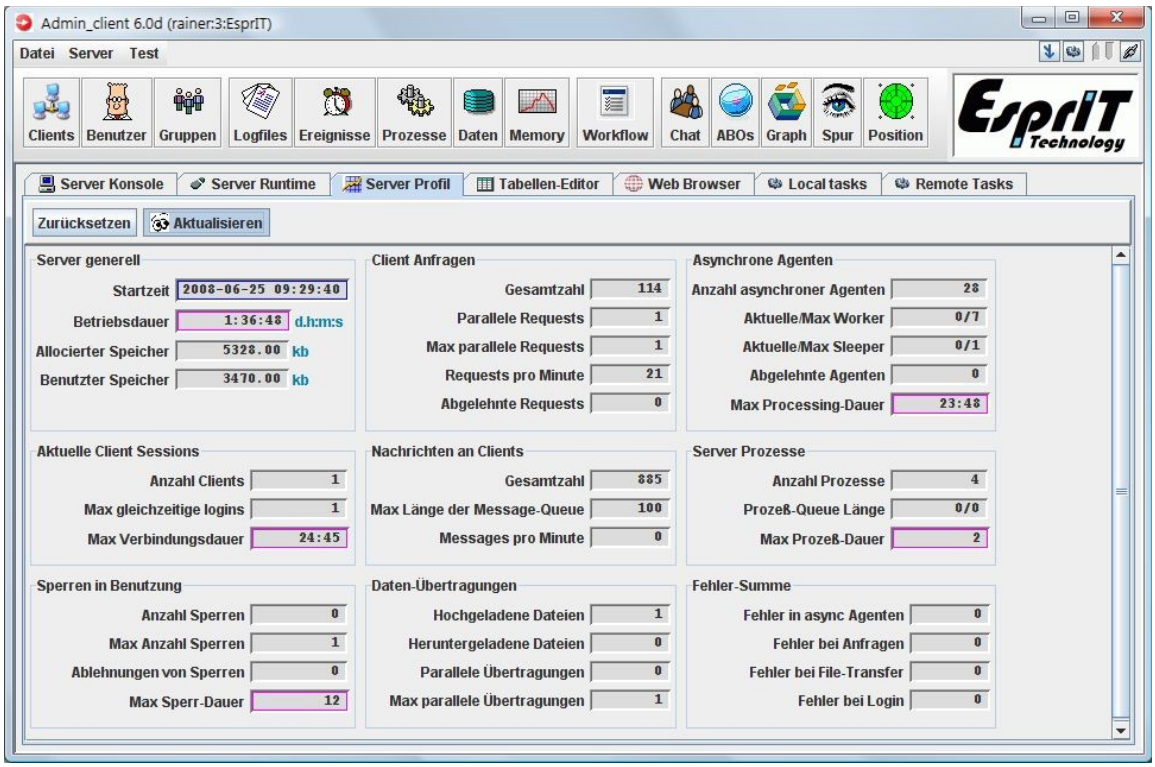


Bild 7) *Das Statistik-Panel gibt eine Übersicht über die wichtigsten Zustandsparameter des Servers.*

9.6.2 Resource-Sperren

Der *Esprit*-Server bietet dem Client die Möglichkeit sog. *Resource-Sperren* anzufordern. Möchte der Client z.B. eine Serverdatei lesen, dann kann er eine *shared* Sperre dafür belegen, die sicherstellt, dass die Datei nicht gelöscht werden kann, solange sie im Zugriff ist. Andere Clients können dieselbe Datei gleichzeitig ebenfalls mit *shared* Sperren belegen.

Möchte der Client die Datei editieren, dann öffnet er sie mit einer *exklusiven* Sperre. Da eine derartige Sperre nur einmal existieren kann ist damit sichergestellt, dass kein anderer diese Datei gleichzeitig verändern kann. Der rein lesende Zugriff von anderen Benutzern ist nach wie vor möglich.

Resource-Sperren werden vom Server besonders sorgfältig verwaltet. Es darf z.B. niemals passieren, dass eine Sperre für immer belegt bleibt, nur weil ein Client „vergisst“, sie aufzuheben. Deshalb sind Sperren „geleased“, d. h. der Client muss stets von neuem sein Interesse bekunden, indem er die Sperre regelmäßig triggert. Sollte dies nicht mehr geschehen, löst der Server die Sperre automatisch auf. Um all das muss sich der Programmierer nicht kümmern, da es „unter der Haube“ automatisch geschieht.

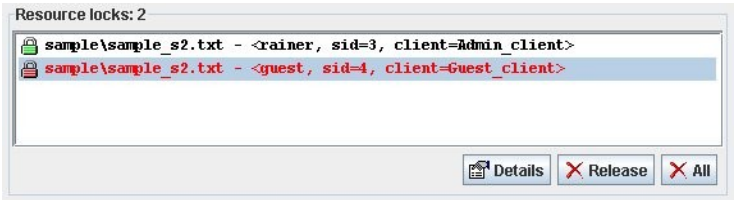


Bild 8) *Diese Liste zeigt die gerade belegten Resource-Sperren. Der Administrator kann die Freigabe erzwingen.*

Ein Client, der eine Sperre sehr lange belegt, erhält automatisch nach einem gewissen Timeout eine Warnung, die ihn auffordert, die Sperre möglichst bald wieder freizugeben. Der Administrator kann alle aktiven Sperren in einer Liste sehen und hat stets die Möglichkeit, sie zwangsweise aufzuheben.

Wenn ein Client sich abmeldet werden alle seine Sperren ohnehin beseitigt.

Sperren sind im Übrigen nicht auf Dateien beschränkt. Der Mechanismus ist sehr allgemein und kann sich auf beliebige Server-Objekte beziehen. So kann z.B. eine Datenbankverbindung ebenfalls mit einer Sperre belegt werden.

9.6.3 Verwaltung remoter Tasks

Remote Tasks erledigen asynchrone Aufgaben für Clients. Ein Client kann den Server im Prinzip mit vielen Remoten Tasks beschäftigen, die all unterschiedlich lange leben und/oder den Server unterschiedlich stark belasten (siehe: *Worker* und *Sleeper* Tasks). Der Server die Fortschrittsmeldungen von Remoten Tasks an den Client, so dass dieser die Aktivität beobachten kann. Der Administrator kann diese Aktivität beobachten und ggf. den Abbruch bestimmter Tasks erzwingen.

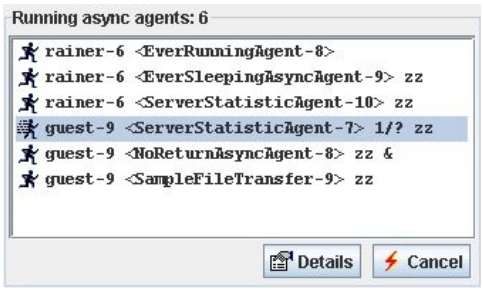


Bild 9) Diese Liste zeigt die gerade laufenden asynchronen Tasks.

9.6.4 Laufende File-Transfers

Prinzipiell erlaubt der Server das parallele Laufen mehrerer File-Transfers von mehreren Benutzern. Da sie über den Transfer-Kanal abgewickelt werden und i.d.R. mit niedriger Priorität laufen, stören sie den Normalbetrieb des Servers nicht sonderlich. Nichtsdestotrotz kann der Administrator eine maximale Anzahl vorgeben damit die Netzwerkbelastung nicht zu hoch wird. Das Admin-Tool zeigt die gerade laufenden Transfers an und ermöglicht dem Administrator auch den Abbruch, falls erforderlich.

Bild 10) Diese Liste zeigt die gerade laufenden File-Transfers.

9.6.5 Email-Benachrichtigung

Im Server können Ereignisse passieren, die dem Administrator nicht verborgen bleiben dürfen. Wenn z.B. ein *ServerBatchTask* auf einen Fehler gelaufen ist, kann das evtl. schlimme Konsequenzen haben. Für solche Fälle kann man Mail-Adressen von Personen hinterlegen, die benachrichtigt werden sollen. Sie erhalten dann eine Email mit einer detaillierten Fehlerbeschreibung. Welche Ereignisse eine Email-Benachrichtigung auslösen ist frei programmierbar.

9.6.6 Session-Liste

In der Session-Liste des Admin-Tools werden alle verbundenen Clients angezeigt. Aus dieser Liste heraus kann der Administrator die Clients individuell „ansprechen“ indem er ihnen sogenannte *DownRequests* schickt. Er könnte z.B. einem Client eine Administrator-Nachricht zukommen lassen

oder ihn zwingen alle seine Aktivitäten auf dem Server (laufende Remote Tasks) abzubrechen. Natürlich kann er ihn auch zwangsweise ausloggen oder gar terminieren.

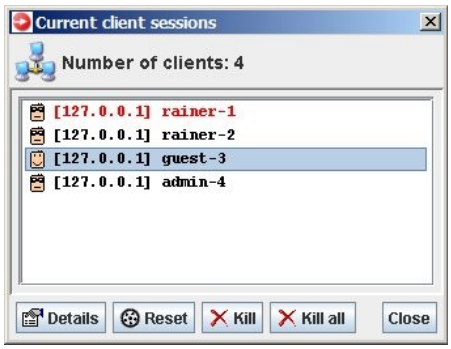


Bild 11) Die Session-Liste zeigt alle verbundenen Clients

Normalerweise ist der Client in der Liste mit seinem Benutzernamen eingetragen. Es gibt aber auch Client-Sessions, die ohne Authentifizierung laufen, d.h. es ist gar kein Benutzer eingeloggt. Solche Sessions haben nur sehr eingeschränkte Rechte; man erkennt sie daran, dass lediglich die Session-ID ohne Benutzername angezeigt wird.

9.6.7 Co-Server Verbindungen

Ein *Esprit*-Server kann sich als sog. *Co-Client* mit einem anderen *Esprit*-Server (seinem *Co-Server*) verbinden. Eine derartige Verbindung ist stets unauthentifiziert (ohne Benutzer-Login), aber trotzdem nicht anonym, denn statt eines Benutzernamens wird der jeweils eindeutige Servername zur Identifizierung herangezogen. Diese Verbindung hat immer alle Rechte.

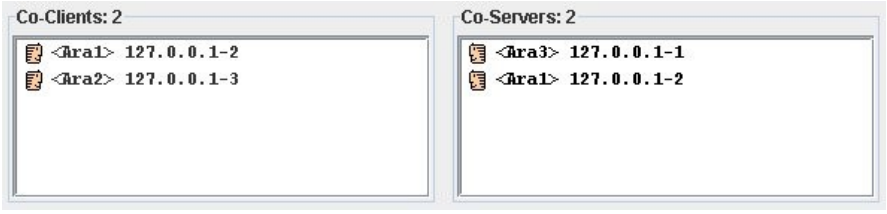


Bild 12) Diese Listen zeigen, mit welchen Co-Servern der betreffende Server verbunden ist, bzw. welches seine Co-Clients sind.

Sollte ein *Co-Server* heruntergefahren werden (oder die Verbindung abreißen), dann wird der betreffende *Co-Client* ständig versuchen, die Verbindung wieder aufzubauen, er erscheint in der Liste solange ausgegraut (siehe: Server Kaskadierung).

9.7 Benutzer-Verwaltung

Kein Server kommt ohne Benutzerverwaltung aus. Im *Esprit*-Server wird dies bewerkstelligt von einer *UserManager* Instanz, die Benutzereinträge wahlweise in einer Serverdatei oder in einer Datenbank verwaltet. Mit dem Admin-Client können Benutzer angelegt, geändert oder gelöscht werden. Alle Änderungen werden – nach Esprit Manier – an andere Clients kommuniziert und sind von allen sofort sichtbar.

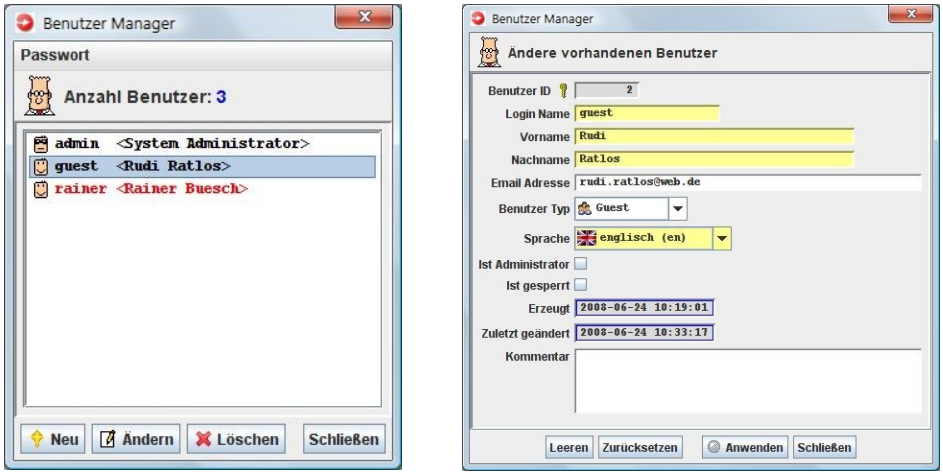


Bild 13) Der UserManager unterstützt Sie bei der Verwaltung Ihrer Benutzer

9.7.1 Passwort Vergabe

Beim Anlegen eines Benutzers erhält dieser automatisch ein vom Administrator vorgegebenes Passwort. Bei seinem ersten Login kann der Benutzer sich ein eigenes Passwort vergeben, welches auf dem Server in verschlüsselter Form gespeichert wird, es ist also auch für den Administrator nicht lesbar. Sollte ein Benutzer sein Passwort vergessen haben, kann der Administrator helfen indem er ihm das ursprüngliche Standard-Passwort wieder zuweist.

9.7.2 Benutzer Login

Der *Esprit*-Server unterscheidet zwischen einer Client-Session und einer Benutzer-authentifizierten Session. Wenn der Benutzer den Login-Dialog sieht, ist bereits eine Client-Session aktiv, d.h. es existiert bereits eine Verbindung zum Server. Nach korrekter Eingabe von Benutzername und Passwort ist diese Session dann authentifiziert. Ein Benutzer kann sich ausloggen und als jemand Anderer wieder einloggen, ohne die Serververbindung zu trennen. Sowohl die Login-Vorgänge, als auch die Verbindungs-Auf/Abbauvorgänge von Clients werden per Nachricht an alle anderen Clients kommuniziert. Daher ist jedem sofort bekannt, wer sonst noch online ist - die Voraussetzung für eine zuverlässige und schnelle Inter-Client-Kommunikation.

9.7.3 Kundenspezifische Benutzerverwaltung

Typischerweise sind Benutzerverwaltungen sehr kundenspezifisch. Deshalb können Kunden über eine vorgesehene Schnittstelle Ihre eigene Implementierung eines *UserManagers* einstöpseln. Dazu stellen Sie eine eigene Klasse *MyUserManager* bereit (abgeleitet von *UserManager*), in der bestimmte Methoden so überschrieben sind, dass die Benutzereinträge z.B. in Ihrer Datenbank gespeichert werden. Die Clients „merken“ davon nichts, da sich an ihrer Schnittstelle ja nichts ändert.

10 Architektur des Esprit Clients

Dieses Kapitel bietet einen Überblick über die wichtigsten Funktionen des *Esprit*-Clients. Ein spezieller Administrations-Client ist in der Software enthalten. Beliebige andere Clients können relativ schnell und einfach entwickelt werden, da die *Esprit* Software dazu baukastenartige Komponenten bereitstellt.

10.1 Verbindung zum Server

Sämtliche Client-Grundfunktionalität ist gekapselt in der eingangs erwähnten *ClientContext* Klasse.

Sie bietet Zugriff auf alle enthaltenen Client-Komponenten. Die Instantiierung dieser Klasse entspricht der Verbindungsaufnahme zum *Esprit*-Server. Ein Client ist typischerweise eine Kundenspezifische Entwicklung, d.h. in Wirklichkeit instantiieren Sie die My*ClientContext* Klasse, abgeleitet von *ClientContext*, die Ihre spezifische Funktionalität beinhaltet.

10.1.1 Software Update

Bei der Verbindungsaufnahme macht der Client mit dem Server als erstes einen Versionenabgleich. Wird dabei festgestellt, dass die Softwareversion des Clients nicht mehr aktuell ist, geschieht zunächst ein Download der neuesten Software. Der Benutzer merkt davon nichts.

➔ Hierbei kommt die Java Webstart Technologie zur Anwendung, die es ermöglicht, dass der einst erhebliche Aufwand für die Softwareinstallation der Clients vollständig entfällt.

10.1.2 Login

Erst nachdem die Verbindung zum Server bereits besteht, erscheint der Login-Dialog, der bei korrekter Eingabe von Benutzernamen und Passwort die Verbindung authentifiziert. Der Login-Request liefert alle Daten, die der Client für seine vollständige Initialisierung benötigt.

Bild 14) Der Login-Dialog des Esprit-Clients

10.1.3 Reconnect

Da der Client sich die einmal eingegebenen Login-Daten merkt, kann er jederzeit per Mausklick offline und wieder online geschaltet werden. Sollte die Verbindung unabsichtlich abreißen, z.B. wegen Netzwerkproblemen, dann versucht er periodisch, sie wieder aufzubauen. Gelingt dies, führt er eine vollständige Reinitialisierung durch, denn es könnte ja sein, dass er wichtige Nachrichten verpasst hat, während er offline war.

10.2 Message Empfänger

➔ Eine der wichtigsten Komponenten des *Esprit*-Clients ist der Message-Empfänger, über den die vom Server kommenden Nachrichten empfangen und verarbeitet werden. Die Nachrichten sind typisiert, um sie je nach Typ an verschiedene Client-Module delegieren zu können. Hier eine Übersicht über die wichtigsten Typen für die man sich beim *ClientConnector* mit Hilfe der entsprechenden `addXxxListener(...)` Methoden registrieren kann:

➔ **Kundenspezifische Nachrichten**
`addNetEventListener(...)`
Eine kundenspezifische Client-Implementierung kann beliebige Nachrichten selbst erfinden Beispiel: das oben beschriebene *PositionEvent*.

➔ **Fortschrittsmeldungen von Hintergrundprozessen und remoten Tasks**
`addServerBatchTaskListener(...)`
`addRemoteTaskListener(...)`
Der Server meldet Zustandsänderungen von serverseitig laufenden asynchronen Prozessen per *ServerBatchTaskEvents* bzw. *RemoteTaskEvents*. Dadurch können Clients u.a. die Prozess-Fortschritte in einem Laufbalken visualisieren.

➔ **Logmeldungen des Servers**
`addServerLogMessageListener(...)`
Der Server schreibt Logmeldungen nicht nur in seine Logdatei(en) sondern sendet sie auch per *LogMessageEvent* an registrierte Clients, wo der Benutzer sie direkt in einer Konsole verfolgen kann.

- **Meldungen über Datensatzänderungen**
  addRemoteRecordChangeListener(...)
  In der Datenbank geänderte Datensätze werden als *RemoteRecordChangeEvents* in einer Nachricht verschickt, woraufhin die Clients ihre lokale Sicht aktualisieren ohne selbst eine Datenbank-Anfrage machen zu müssen.
- **EMI Nachricht**
  addEmiMessageListener(...)
  In einer solchen Nachricht erhält der Client die asynchrone Antwort auf eine von ihm gestellte EMI-Anfrage.
- **Nachricht von einem anderen Client**
  Ein Client kann per Anfrage eine Nachricht an sich selbst, einen anderen Client, an bestimmte Benutzergruppen oder an alle eingeloggten Clients senden. Auf diesem Wege können Clients auch direkt miteinander kommunizieren.
- **DownRequests (Kommandos an Clients)**
  Eine Nachricht kann ein Kommando beinhalten, welches der Client dann ausführt (*DownRequest*).

Sofern der Client nicht selbst der Verursacher einer Nachricht ist, erhält er Server-Nachrichten eines bestimmten Typs nur dann, wenn er sich explizit dafür registriert hat. So bekommt z.B. nur der Administrator-Client die Logmeldungen des Servers zugeschickt. Es gibt andererseits auch spezielle systeminterne Nachrichten, deren Empfang man nicht unterbinden kann.

10.3 Schnittstelle zu Betriebssystemprozessen

Der Client beinhaltet eine Schnittstelle, über die andere Prozesse aus dem Betriebssystem aufgerufen werden können, sei es ein Texteditor, ein Spreadsheet-Werkzeug oder gar ein CAD-System. Der Prozess läuft dann unter der Kontrolle des Clients. Auf diese Weise lassen sich „Fremdwerkzeuge“ einbinden, dessen erzeugte Daten der Client dann verarbeiten kann. Beispielsweise könnte mit einem CAD-System eine Input-Datei für eine Finite-Elemente-Berechnung angelegt werden, die dann beim Verlassen des Werkzeugs automatisch zum Server hochgeladen und dort weiterverarbeitet wird.

10.4 Steuerung von Clients (mit Down-Requests)

Ein Client (oder der Server selbst) kann einem anderen Client eine Nachricht schicken, die ein *DownRequest* Objekt enthält. Ein Down-Request ist ein Kommando, welches der Client dann auszuführen hat.

Per *DownRequest* kann der Server den Client praktisch steuern, eine Option, die viele Möglichkeiten eröffnet. Beispiele dafür sind der *ClientResetDownRequest*, welcher den Client veranlaßt, alle asynchronen Aktivitäten einzustellen, oder der *ClientKillDownRequest*, mit dem der Administrator einen Client zwangsweise terminiert.

10.4.1 Synchrone Down-Requests

Down-Requests sind standardmäßig synchron. Das heißt: der Initiator des Down-Request wartet solange, bis er von dem angesprochenen Client dessen Down-Response erhalten hat. Diese enthält dann entweder ein Ergebnis-Objekt oder einen Fehler.

Zur Begrenzung der Wartezeit kann im *DownRequest* ein Timeout Wert eingestellt werden. Wird dieser überschritten, so schlägt der Request mit einer *TimeoutException* fehl.

10.4.2 Down-Requests an viele Clients

Wird bei einem *Down-Request* keine Ziel-Session angegeben, dann geht er an **alle** Clients. Die Antwort besteht dann aus einer Liste der *Down-Responses* aller Clients. Auf diese Weise ist es sehr einfach, Informationen von allen Clients einzusammeln.

Anwendungsbeispiele wären das Abfragen der Konfiguration der Client-Rechner oder die Kontrolle über deren Software-Installation.

10.4.3 Asynchrone Down-Requests

Als weiteres Beispiel könnte der Server den Client veranlassen, eine Ergebnisdatei abzuholen, die soeben auf Serverseite fertig geworden ist. *DownRequests* können synchron oder asynchron abgearbeitet werden. Im letzten Beispiel wäre eine asynchrone Ausführung angebracht, damit der Veranlasser des Down-Requests nicht auf den Datei-Download warten muss.

➔ *DownRequests* ermöglichen, dass der Client auf eine Nachricht hin aktiv wird. Sie erlauben es, den Client vom Server aus zu steuern oder Informationen von ihm zu erfragen.

10.5 Client Aktivitäts-Monitor

Ein Client macht ggf. viele Dinge gleichzeitig: er hat mehrere Remote Tasks gestartet, überträgt Dateien an den Server, lässt lokale Betriebssystemprozesse laufen und hält gleichzeitig noch ein paar Server-Sperren, die er ständig retriggern muss. Sämtliche Aktivitäten lassen sich im Aktivitäts-Monitor des Clients beobachten und – falls gewünscht – auch abbrechen. Die runden Leuchtdioden markieren Rubriken, in denen Aktivität vorhanden ist, die pfeilartigen Leuchtdioden im *ToolBar* des *MainFrame* zeigen ausgehende Requests bzw. eingehende Messages an.

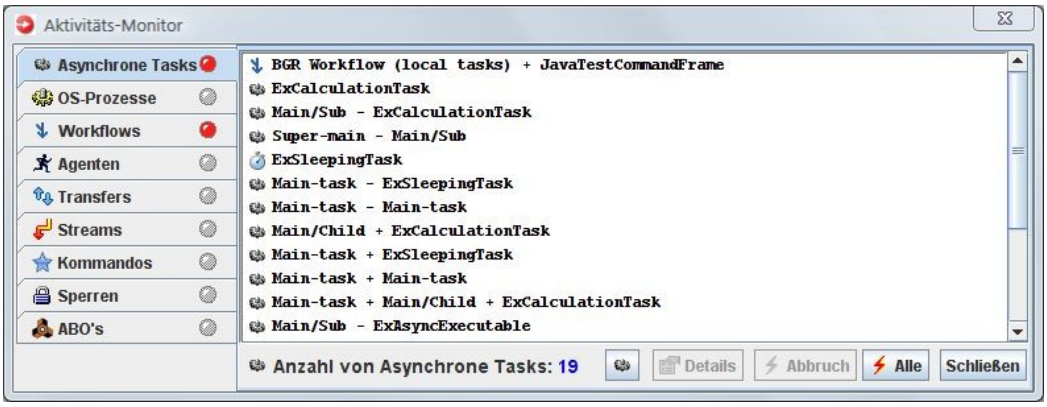


Bild 15) Der Aktivitäts-Monitor zeigt die aktuellen asynchronen Client Aktionen

11 Workflows

Eine der wesentlichen Stärken der *Esprit-Technologie* ist, dass eine Reihe vordefinierter Aufgaben asynchron abgearbeitet werden können – und dies auf eine robuste, beobachtbare Art.

11.1 Ablauf von Workflows

Der folgende beispielhafte prozedurale Ablauf mag dies verdeutlichen:

➔ Client startet CAD System und erzeugt Input-Datei

➔ Input-Datei wird zum Zentral-Server hochgeladen

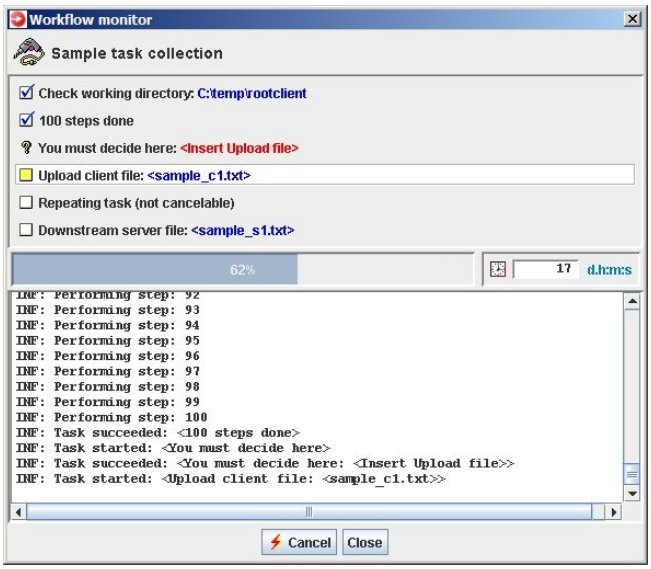
➔ Zentral-Server macht Datenbankeintrag und überträgt den Input weiter an Compute-Server

➔ Compute-Server startet Berechnung und überträgt Zwischenergebnis an Zentral-Server

➔ Zentral-Server macht Datenbankeintrag und meldet Ergebnis an wartenden Client

➔ Client holt Ergebnisdatei von Zentral-Server ab

➔ Client startet CAD-System zur Darstellung des Ergebnisses



Derartige Abläufe lassen sich in einfacher Weise als sog. **Workflows** aufbauen. Ein Workflow ist eine Ansammlung von Aufgaben (Tasks) die asynchron nacheinander ablaufen wobei deren Fortschritt in einem *Workflow-Monitor* visuell verfolgt werden kann. Der Ablauf kann jederzeit vom Benutzer abgebrochen werden (was softwaretechnisch in der Entwicklung eine nicht zu unterschätzende Herausforderung ist).

Fertiggestellte Tasks werden im *Workflow-Monitor* als erledigt abgehakt. Bei Abbruch oder Fehler ist die betreffende Task entsprechend sichtbar markiert.

Bild 16) In Workflows lassen sich alle erdenklichen synchronen und asynchronen Aktionen einbinden, auch solche, die nicht lokal auf dem Client, sondern remote auf dem Server ablaufen.

Häufig ist es erforderlich, dass der Benutzer in Abhängigkeit von Zwischenergebnissen entscheiden muss, ob oder wie der *Workflow* fortgesetzt werden soll. Eventuell können Tasks übersprungen oder es müssen neue eingefügt werden. Diese Funktionalität bieten spezielle *Decision-Tasks*, die dem Benutzer per Dialog die möglichen Optionen zur Auswahl anbieten. Der *Workflow* ist also keine statische Taskliste, sondern benutzergesteuert zur Laufzeit veränderbar.

In der im *Workflow-Monitor* eingebauten Konsole werden die Logging-Meldungen des jeweils laufenden Tasks ausgegeben, so dass der Benutzer stets sieht, was gerade passiert.

11.2 Parallel laufende Workflows

In der Praxis können Workflows ggf. stundenlang laufende Prozesse sein. Der Benutzer kann das Workflow-Monitor-Fenster jederzeit schließen und bei Bedarf wieder aus der aktuellen Workflow-Liste hervorholen, um den Stand der Dinge zu kontrollieren. Ist ein Workflow abgelaufen, wird der Monitor automatisch geöffnet und zeigt das Ergebnis an.

Es können beliebig viele unterschiedliche Workflows parallel gestartet werden. Sie alle werden in der Workflow-Liste angezeigt und können von dort aus kontrolliert werden.

11.3 Serverseitige Workflows

Workflows sind so allgemein konzipiert, dass sie auch auf Serverseite laufen können – mit einer Einschränkung: da ein Server keine Benutzeroberfläche besitzt, darf der Workflow keine Tasks beinhalten, die Benutzerinteraktion erfordern. Ein serverseitig laufender Workflow kann vom Client per Workflow-Monitor auf die gleiche Weise beobachtet werden, wie ein lokal Laufender.

➔ Workflows bauen auf dem *Esprit AsyncTask Framework* auf. Siehe dazu:
http://esprit-systems.de/downloads/esprit/docu/EspritAsyncTaskFrameworkDE.pdf

➔ Das Workflow Konzept für lokale und remote Workflows ist in folgendem Dokument beschrieben: http://esprit-systems.de/downloads/esprit/docu/EspritWorkflowProcessingDE.pdf

12 Fazit

Die *Esprit*-Technologie eröffnet eine Vielfalt von Möglichkeiten, die mit bisher üblichen Client/Server Systemen (z.B. Applikations-Server) nur sehr schwer oder gar nicht realisierbar sind. Durch den integrierten Message-Service kann das MVC Prinzip über Netz realisiert werden. Esprit-Clients weisen eine bisher ungekannte Dynamik auf und sind datenmäßig stets auf dem aktuellsten Stand.

12.1 Client/Server Lastverteilung

Da Clients passiv über Änderungen informiert werden, brauchen sie selbst keine unnötigen Anfragen mehr an den Server zu stellen. Jegliches Server-Polling entfällt vollständig. Da Nachrichten fast immer aus kleinen Delta-Informationen bestehen, ist die Performance außerordentlich gut. Im Endeffekt ist die Serverlast relativ gering, wohingegen die Client-Power mehr als sonst üblich, genutzt wird. Im Endeffekt erhält man eine bessere Lastverteilung zwischen Client und Server (siehe: Parallelprozessing auf Client und Server).

12.2 Komplexe prozedurale Abläufe

Ein unschätzbarer Vorteil liegt darin, dass sich synchrone und asynchrone Vorgänge (Remote Tasks, *DownRequests*, *FileTransfers*) in Form von Workflows sehr leicht zu komplexen Prozessabläufen koppeln lassen, in denen viele Rechner, Clients wie Server, involviert sind.

➔ Durch Bündelung synchroner und asynchroner Operationen als Workflows sind komplexe prozedurale Abläufe und über mehrere Rechner verteilte Operationen auf robuste Weise realisierbar.

12.3 EMI als Programmierschnittstelle

Das Konzept von EMI, welches auf der Definition von EPI-Interfaces basiert, macht es extrem einfach, verteilte Logik zu implementieren. Die Netzwerkschicht bleibt dem Programmierer vollständig verborgen. Er kann sich voll auf seine Programmlogik konzentrieren und seine Software auf einer robusten Grundlage aufbauen.

➔ Die Entwicklung einer Anwendung besteht praktisch darin, die erforderlichen EPI-Interfaces zu definieren und zu implementieren.

12.4 Echtes MVC über Netz

Der *Esprit*-Server unterstützt das Model-View-Controller Prinzip (MVC) über Netz. Das heißt: Server-Modelle, die von einem Client verändert werden schicken ihrerseits Events an alle anderen Clients, die daraufhin ihre jeweilige Sicht aktualisieren – und dies mit bestmöglicher Performance!

➔ Die Unterstützung von MVC über Netz ermöglicht die Realisierung von hochdynamischen Clients.

12.5 “Lebende” Business-Objekte

Eine wirkliche Neuheit bilden die sog. *Alive Business Objects*, die auf dem Server global in einem Server-Cache zur Verfügung stehen. Alle Clients, die solche Objekte benutzen, werden automatisch über Änderungen informiert, solange sie Zugriff darauf haben. Damit gibt es für alle Clients stets genau eine Sicht der Welt.

➔ Alive Business Objects sind besonders leicht zu entwickeln und sehr einfach und robust in der Anwendung.

12.6 Rich-Client oder Web-Client?

Der *Esprit*-Technologie ist insbesondere für die Unterstützung sog. Rich-Clients geeignet. Letztere haben gegenüber den Web-Clients den Vorteil der wesentlich leichter handhabbaren Komplexität und besseren Performance, da der Overhead eines Web-Browsers und die damit verbundenen Schwierigkeiten entfallen. Als Nachteil wurde bisher die Notwendigkeit gesehen, dass die Software auf jedem einzelnen Client installiert werden muss. Seit dies aber von Java-Webstart automatisch erledigt wird, dürfte die Diskussion, ob Web- oder Rich-Client, sich neu entfachen.

➔ Der Esprit-Server bietet sich als Standardserver für Rich-Clients an, ähnlich dem Webserver für Web-Clients.

13 Weitere Informationen

Veröffentlichung über Java Datenbank Persistenz mit DBObjects (Java-Spektrum 04/2003)

http://www.sigs.de/publications/js/2003/04/buesch_JS_04_03.pdf

Veröffentlichung über Netzwerkprogrammierung (Java-Magazin 07/2005)

http://www.esprit-systems.de/downloads/esprit/EspritArticleJavaMAGAZIN.pdf

Dokumentation über das *Esprit*-Framework finden Sie auf der Downloadseite von

http://www.esprit-systems.de