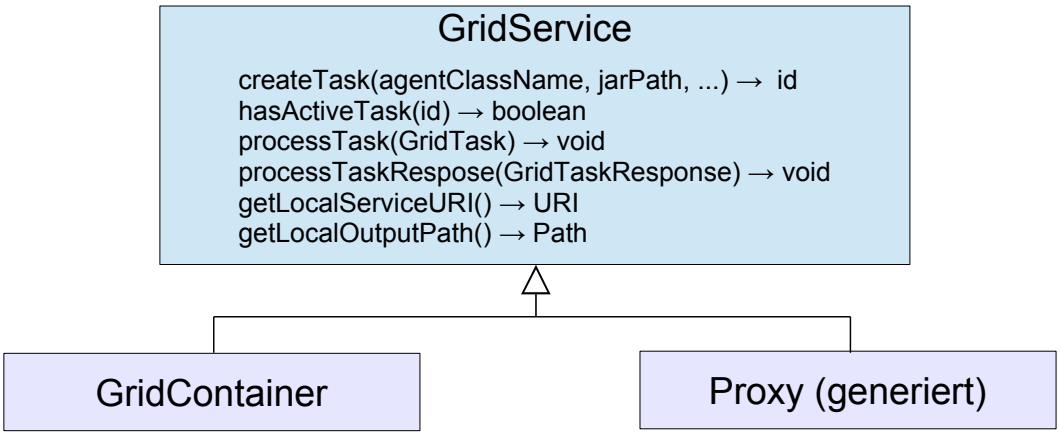
Verteilte Systeme

Übung F - Gruppenarbeit für 2er Gruppen

Bearbeitungszeit 2 Wochen

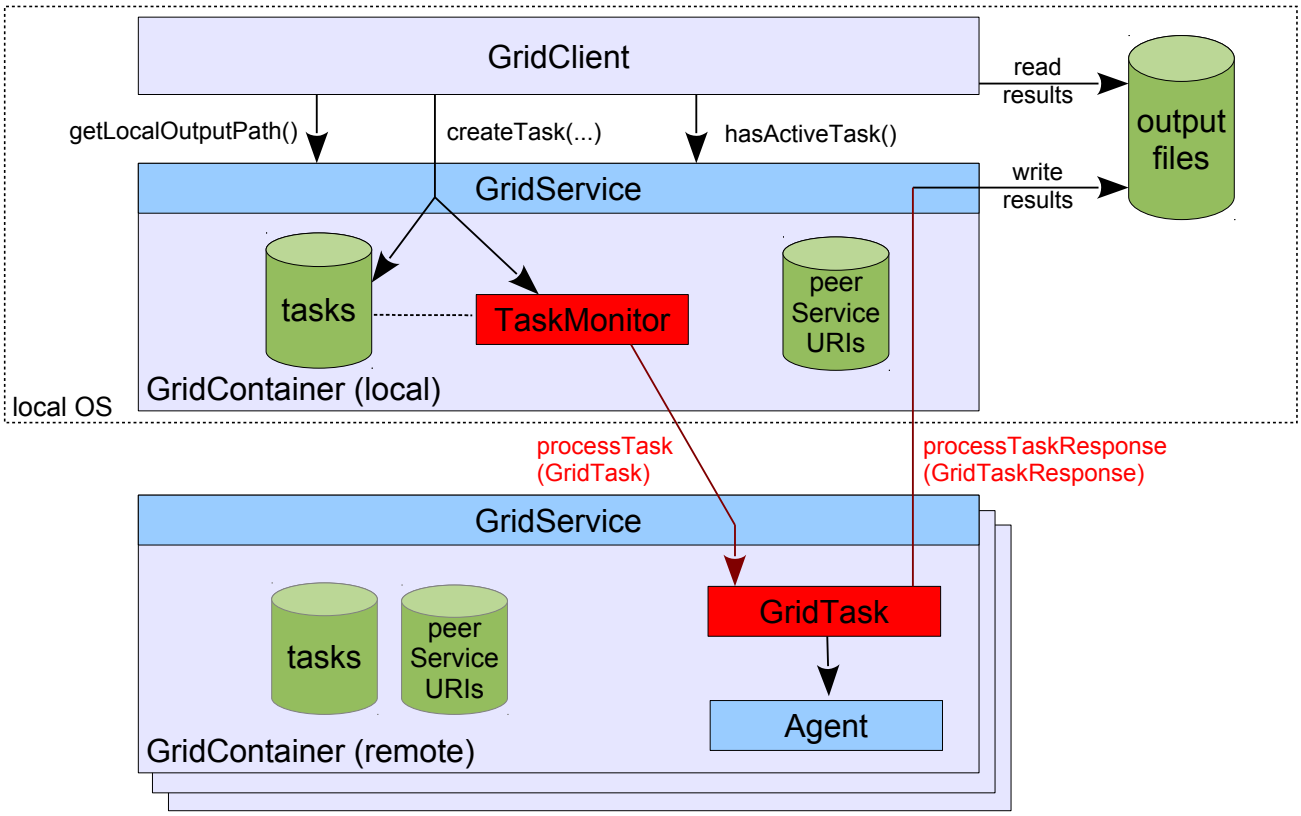
Ziel dieser Übung ist die Erstellung eines **P2P Grid-Containers**. Dieser soll automatisches Deployment von Agenten beim Aufruf, sowie das automatische Undeployment nach Ausführung derselben unterstützen. Die Container-Prozesse stellen dabei RPC-basierte **Web-Services** zur Verfügung; diese werden **bottom-up** mittels JAX-WS generiert:



Die **GridContainer**-Instanzen sind in der Lage beliebige Arbeitspakete zu verteilen bzw. auszuführen. Arbeitspakete werden dabei mittels **GridTask** Instanzen beschrieben, welche im Prinzip *Runnables* darstellen die alle zu ihrer (möglicherweise wiederholten) Ausführung notwendigen Informationen beinhalten. Diese beinhalten daher neben Binärdaten (ein Datei-Pfad) und Properties (einfache Daten) den auszuführenden Code. Dieser wird durch den Namen einer Klasse beschrieben die das **Agent**-Interface implementieren muss, sowie optional durch den Pfad auf ein JAR-File welches den Code dieser Klasse enthält (falls sie nicht Teil des Class-Path beim Container-Start ist).

GridTasks werden nach ihrer Erzeugung über eine generierte Identität identifiziert, womit es möglich wird beim Empfang einer **GridTaskResponse** auf den verursachenden GridTask zu schließen, und das obwohl sowohl die Verteilung der Tasks, als auch die Rücksendung der Ergebnisse aus Lastgründen asynchron (i.e. ONE-WAY in JAX-WS Terminologie) erfolgen muss.

Folgende Abbildung zeigt die fundamentalen Daten und Kommunikationsströme des Grid-Systems; Asynchrone Methoden und Threads sind dabei in Rot gehalten, Prozesse in Grau, Interfaces in Blau, und Daten in Grün:



Beschreibung des System-Designs:

GridClient-Instanzen werden als separate Prozesse neben einer lokalen **GridContainer**-Instanz gestartet, welche wiederum die Service-URIs weiterer „entfernter“ *GridContainer*-Peers kennen muss. Die lokale *GridContainer*-Instanz dient dem *GridClient* dabei als Router und Koordinator. Die *GridClients* erzeugen mittels `GridService.createTask()` neue *GridTasks*, und erhalten als Resultat eine eindeutige **Task-ID** vom lokalen *GridContainer* zurück. Damit können sie mittels Polling, i.e. wiederholte Aufrufe von `GridService.hasActiveTask()`, feststellen wann ein *GridTask* beendet, und daher ein Task-Ergebnis im Ausgabeverzeichnis des *GridContainers* zu erwarten ist.

Im lokalen *GridContainer* wird jeder Task von einem eigenen TaskMonitor beobachtet, welcher bei der Task-Erzeugung im Monitor-Scheduler des Containers für wiederholte Ausführung registriert wird. Ein Task wird durch diesen einmal sofort nach Erzeugung, und dann nach jedem Timeout mittels *GridService.processTask()* an einen zufällig ausgewählten Peer zur asynchronen Ausführung versendet.

Im beauftragten *GridContainer*-Peer wird der Task (ein Clone des Originals!) dann ausgeführt. Dazu muss ein eventuell übertragenes JAR-File temporär gespeichert, und der Task als neuer Thread mit um das JAR-File erweitertem Class-Path gestartet werden. Im Task-Thread wiederum muss eine Agenten-Instanz mittels **Java Reflection API** aus dem Klassennamen erzeugt, diese mit den Task-Daten zusammen ausgeführt, und das Ergebnis als **GridTaskResponse** mittels `GridService.processTaskResponse()` asynchron an den Originator-Service zurück gesendet werden. Falls dies nicht gelingt weil der Originator-Service inzwischen offline ist, muss auch dies einige Male wiederholt werden.

Glückt die Übertragung, dann wird das Ausführungsergebnis im Ausgabe-Verzeichnis des verteilenden *GridContainers* als Datei gespeichert, wobei sich der Dateiname aus der Task-ID sowie einer Agent-spezifischen Namenserweiterung zusammensetzt. Sind alle vom *GridClient* erwarteten Ausführungsergebnisse eingetroffen, kann dieser ein Gesamtergebnis zusammensetzen und die Prozessierung erfolgreich beenden.

Aufgabe 1: Setup, Service-Interface & Halo Annotationen

Benennt die Klassen *GridTaskSkeleton*, *GridServiceSkeleton* und *GridContainerSkeleton* in **GridTask**, **GridService** und **GridContainer** um. Damit sollten alle Compilefehler im Packet `de.htw.ds.grid` verschwinden. Kopiert zudem die Datei `Baumeister/Verteilte Systeme/code/distributed-systems-plugins.jar` in Euer lokales Dateisystem. Diese JAR-Datei enthält die Agenten-Klassen welche zur Ausführung der Beispiel-Clients bzw. Testfälle benötigt werden, der Pfad dieser Datei ist dabei an entsprechender Stelle zu übergeben.

Das Service-Interface ist komplett was Java-Definionen betrifft, und die Container-Klasse ist bereits so definiert dass sie das Service-Interface implementiert. Was fehlt sind jedoch einige JAX-WS und JAX-B Annotationen um die Halo-Objekte korrekt zu marshalen; ohne lässt sich der Container erst gar nicht starten. Fügt daher zum **Service-Interface** passende Annotationen hinzu um

- das Interface an sich als JAX-WS Service-Interface zu deklarieren
- jeden Methoden-Parameter in der WSDL mit sprechenden Parameternamen auszustatten
- die Methoden `processTask()` sowie `processTaskResponse()` für asynchrone Ausführung zu markieren.
- das Resultat von `getOutputPath()`, sowie die zwei Parameter `jarPath` und `dataPath` von `createTask()`, als Werte zu markieren die in der generierten WSDL als Strings auszuweisen sind, und die zur Laufzeit mittels eines Typ-Adapters von *Path* nach *String* gemarshaled, und beim Unmarshaling wieder von *String* zurück nach *Path* gewandelt werden sollen; dies ist notwendig weil Path ein nicht mit @XmlSeeAlso annotiertes Interface, und zudem die Implementierungsklasse (UnixPath, WindowsPath, usw.) plattformspezifisch ist → verwendet dazu an den genannten drei Stellen die Annotation `@XmlJavaTypeAdapter(PathStringAdapter.class)`
- den Parameter `properties` der Methode `createTask()` in der generierten WSDL als *StringMapEntry[]* auszuweisen, und zur Laufzeit von *Map<String,String>* nach *StringMapEntry[]* und zurück zu wandeln; dies ist notwendig weil das Interface Map nicht vom Collection-Interface erbt, und damit beim Marshaling keinerlei automatische Sonderbehandlung erfährt → seht Euch daher die Klasse *PathStringAdapter* genauer an, entwerft analog eine Adapter-Klasse *StringMapAdapter* welche über `map.entrySet()` itertiert um Arrays der vorhandenen Klasse *StringMapEntry* zu erzeugen, und annotiert den Parameter `properties` entsprechend

Annotiert des weiteren in der Halo-Klasse **GridTask**

- die Klasse selbst mittels `@XmlType` damit diese in der WSDL einen eigenen Namespace erhält
- die Klasse selbst mittels `@XmlAccessorType(XmlAccessType.FIELD)` damit nur Instanzvariablen für das Marshaling in Betracht gezogen werden
- das Feld `properties` um dieses in der WSDL als *StringMapEntry[]* auszuweisen und zur Laufzeit passend zu adaptieren (siehe oben)
- die Felder `jarPath` und `dataPath` um diese in der WSDL jeweils als Typ „base64Binary“ auszuweisen; es soll hier also nicht ein Pfad, sondern der binäre Inhalt der Datei auf welche der jeweilige Pfad zeigt übertragen, auf der Gegenseite in eine temporäre Datei gespeichert, und deren Pfad im übertragenen GridTask wieder verankert werden! Annotiert diesen relativ komplexen Vorgang einfach wie oben mittels der vorhandenen Adapterklasse PathContentAdapter
- jedes dazu passende Feld so dass es in der WSDL platzsparend als XML-Attribut statt als XML-Element ausgewiesen wird

Schreibt sodann einen Testcase der einen JAX-WS Proxy für eine lokale GridService-Instanz erzeugt (analog zu `de.htw.ds.chat.RpcChatClient.main()`, siehe auch `GridContainer.selectActivePeer()`), und ruft darin jede Service-Methode mit geeigneten Parametern einmal auf. Setzt im Debugger Unterbrechungspunkte in jeder dieser Methoden, startet eine GridContainer-Instanz im Debug-Modus, und überprüft ob sich

- alle Service-Methoden ohne Exceptions aufrufen lassen
- sowie ob alle Parameter und Resultate vollständig übertragen werden.

Überprüft abschließend mittels eines Web-Browsers die vom gestarteten GridContainer generierte WSDL (Service-URI + „?wsdl“), sowie die dort verlinkte Types-Sektion (Service-URI + „?xsd=1“)

Aufgabe 2: GridTask

Implementiert die mit TODO markierten Teile der Klasse *GridTask* unter Beachtung der dort beschriebenen Funktionalität.

Testet die *GridTask* Klasse indem ihr einen *GridContainer* im Debug-Modus startet, und in der Methode `processTaskResponse()` einen Breakpoint setzt. Startet sodann einen *TestEchoClient* (ist im Paket enthalten) mit der Service-URI eures *GridContainers*. Dieser Client sollte einen *EchoAgent* in einem künstlichen Task prozessieren, und dann eine *GridTaskResponse* an den *GridContainer* senden. Dieser sollte daraufhin auf den Breakpoint auflaufen. Prüft im Debugger ob die übertragene *GridTaskResponse* eine identity != 0, einen dataType = "bin", und einen Verweis auf eine temporäre Datei mit dem Inhalt „ABCDEFG" (binär [64,65,66,67,68,69,70]) aufweist.

Beachtet dabei dass die Klasse *EchoAgent* im Gegensatz zu normalen Agenten in eurem Projekt enthalten, und damit durch einen *GridContainer* „normal" ladbar ist. Hier ist daher kein spezieller Class-Loader nötig um die Agenten-Klasse laden zu können, im Gegensatz zu Aufgabe 3!

Aufgabe 3: GridContainer

Implementiert die mit TODO markierten Teile der Klasse *GridContainer* unter Beachtung der dort beschriebenen Funktionalität.

Testet das Grid, indem ihr drei *GridContainer*-Instanzen startet: Zwei „entfernte" Container als Peers, und einen lokalen Container als P2P-Gegenstück zu lokal ausgeführten Grid-Clients. Der lokale GridContainer muss dabei die Service-URIs der anderen beiden Peers beim Start übergeben bekommen. Die „entfernten" Container können natürlich zum Testen im lokalen OS ausgeführt werden, achtet dann aber darauf dass jeder der drei GridContainer sein eigenes Ausgabeverzeichnis bekommt! Für Performance-Messungen oder gar Betrieb ist ein solcher Setup dagegen natürlich nicht sinnvoll, hier müssen die entfernten Container in separaten Betriebssystem-Instanzen laufen.

- Startet zum Testen die Klasse **GenericSimpleGridClient** mit folgenden Argumenten:

```
<Service-URI>
de.htw.ds.grid.agent.ImageResizeAgent
"<globaler Pfad auf distributed-systems-plugins.jar>"
"<globaler Pfad auf JPEG-Image"
2 30 width=50 height=50
```

  Der gegebene *ImageResizeAgent* befindet sich dabei in *DistributedSystemsPlugins.jar*. Er ist in der Lage die Größe eines gegebenen JPG-Images zu verändern, im Falle der gegebenen Parameter auf 200*200 Pixel. Der Timeout für diese Operation beträgt dabei die gegebenen 60000ms = eine Minute. Überprüft ob sich im Ausgabe-Directory des lokalen Grid-Containers nun ein JPG-Image mit den gegebenen Dimensionen befindet. Eine reale Grid-Anwendung wäre z.B. das massenhafte, verteilte Erzeugen von Thumbnails.

- Startet des Weiteren die Klasse **GenericAudioGridClient** mit folgenden Argumenten:

```
<Service-URI>
de.htw.ds.grid.agent.AudioDoomAgent
"<globaler Pfad auf distributed-systems-plugins.jar>"
"<globaler Pfad auf (stereo) WAV-Audio Quelle"
"<globaler Pfad auf WAV-Audio Ziel"
5 60 2000000
```

  Falls ihr kein passendes WAV-File zur Hand habt, findet sich unter `Baumeister/Verteilte Systeme/Queens Of The Stone Age - Born To Hula (loop).wav` eines welches geeignet ist. Durch den Client wird die Audio-Datei in Abschnitte zu jeweils zwei Millionen Frames aufgeteilt, und dann abschnittsweise durch den *AudioDoomAgent* manipuliert. Überprüft ob sich im Ausgabe-Directory des lokalen Grid-Containers nun eine WAV-Datei mit verdoppelter Abspiellänge und halbierter Tonhöhe befindet. Beobachtet dabei wie das Grid die Abschnitte auf die entfernten *Grid-Container* verteilt! Eine reale Anwendung dieser Vorgehensweise wäre z.B. die Format- und Größenkonvertierung eines Videos, bei Verwendung eines für Video-Frames geeigneten Clients und Agenten!

- Startet schließlich wiederum die Klasse **GenericAudioGridClient** mit folgenden Argumenten:

```
<Service-URI>
de.htw.ds.grid.agent.AudioCompressorAgent
"<globaler Pfad auf distributed-systems-plugins.jar>"
"<globaler Pfad auf (stereo) WAV-Audio Quelle"
"<globaler Pfad auf WAV-Audio Ziel"
5 60 2000000 compressionRatio=3.0
```

  Dadurch wird eine Audio-Datei wiederum in Abschnitte aufgeteilt, und dann abschnittsweise durch einen *AudioCompressorAgent* manipuliert welcher einen nichtlinearen Verstärker (Kompressor bzw. Expander) auf das Audio-Material anwendet. Überprüft ob sich im Ausgabe-Directory des lokalen Grid-Containers nun eine WAV-Datei befindet bei welchem der Lautstärke-Kontrast deutlich erniedrigt ist. Leise Teile werden dabei mehr verstärkt als laute Teile, dadurch klingt die Audio-Datei insgesamt wesentlich lauter und sanft verzerrt.

Wen brennend interessiert wie diese (und viele weitere) Audio-Manipulationen vonstatten gehen, der belege das AWE-Fach *Virtuelle Klangwelten* an der HTW ...