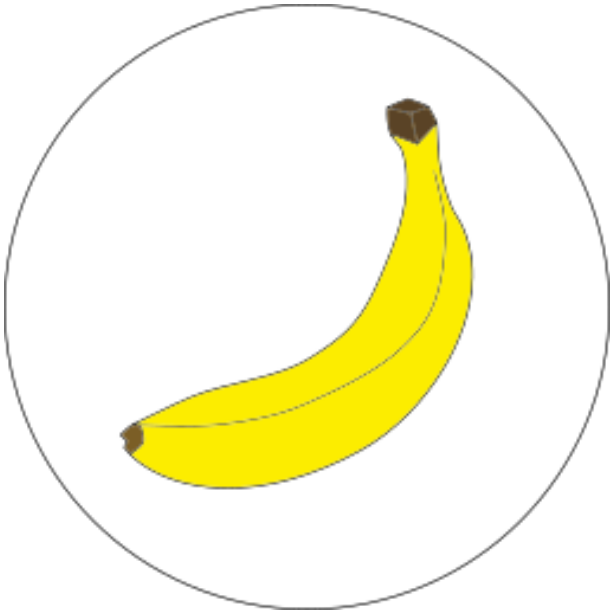


Balanced Banana

A Distributed Task Scheduling System

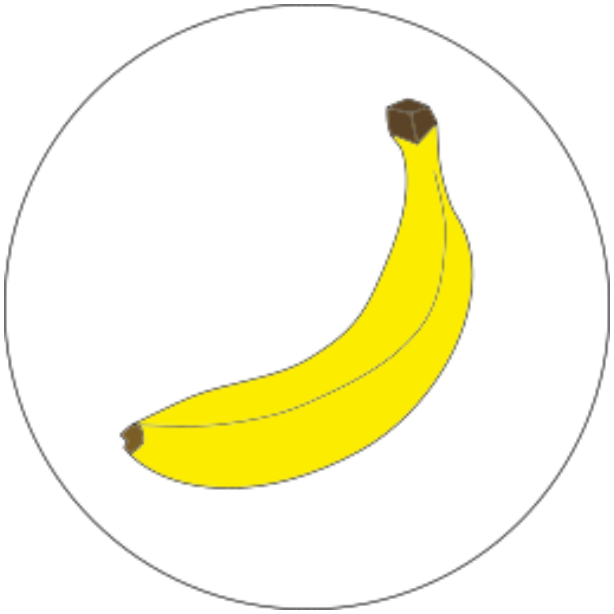
Abschluss

Niklas Lorenz, Thomas Häuselmann, Rakan Zeid Al Masri, Christopher Lukas Homberger und Jonas Seiler

31. März 2020

1 Einleitung

Dieses Dokument enthält alle vorherigen Dokumente, die für unser Praxis der Softwareentwicklung-Projekt, Balanced Banana, ein System zur Verteilung von Aufgaben, erstellt wurden. Wir haben das Wasserfallmodell für den Entwurf und die Implementierung unseres Projekts verwendet.



Balanced Banana

A Distributed Task Scheduling System

Pflichtenheft

Niklas Lorenz, Thomas Häuselmann, Rakan Zeid Al Masri, Christopher Lukas Homberger und Jonas Seiler

8. Januar 2020

Inhaltsverzeichnis

1 Einleitung

Die Verteilung rechenintensiver Aufgaben ist ein in vielen Unternehmen übliches Problem. Wenn ein Team größer wird, so werden auch die verfügbaren Rechenressourcen und die darauf ausgeführten Aufgaben größer und komplexer.

Es wird immer schwieriger, die vorhandenen Ressourcen effizient und gerecht an die verschiedenen Mitarbeiter und Aufgaben zu verteilen. Aktuell stehen individuelle Arbeiter zur Verfügung, die von jedem Mitarbeiter beliebig verwendet werden können. So können Aufgaben nur bearbeitet werden, wenn zum Zeitpunkt der Anfrage ein Arbeiter frei steht. Doch das ist weder fair einem einzelnen Benutzer gegenüber, noch kann die Hardware dadurch gut ausgelastet werden. Wird eine Aufgabe beispielsweise mitten in der Nacht abgeschlossen, ist niemand da, um eine neue Aufgaben ausführen zu lassen.

Es gibt auf dem Markt bereits viele Lösungen für dieses Problem, allerdings sind sie sehr komplex und nicht leicht erweiterbar. Balanced Banana löst dieses Problem maßgeschneidert an die Bedürfnisse des CES.

Mit Balanced Banana soll der Benutzer seine Aufgabe unkompliziert von der Befehlszeile abschicken können, sodass diese automatisch auf einen bereitstehenden Arbeiter verteilt wird. Die Aufgaben sind in Docker-Containern von anderen Aufgaben abgetrennt.

Darüber hinaus soll der Benutzer durch zusätzliche Parameter in der Lage sein, weitere Einschränkungen und/oder Bedingungen für seine Aufgaben anzugeben.

Balanced Banana soll in der Lage sein, die anstehenden Aufgaben effizient auf die verfügbaren Arbeiter zu verteilen. Hierbei werden Größe, Priorität und gegebenen Einschränkungen berücksichtigt. Um möglichst flexibel einsetzbar zu sein, soll außerdem die Verteilerlogik über eine feste Schnittstelle jederzeit änderbar sein, sodass Aspekten wie Fairness unterschiedliche Relevanz eingeräumt werden kann.

2 Zielbestimmung

2.1 Kundenvorgaben

- Der Benutzer kann seine Aufgabe unkompliziert von der Befehlszeile abschicken.
- Parameter ermöglichen die Angabe von Einschränkungen und Ansprüchen, wie zum Beispiel Priorität oder minimal benötigter Arbeitsspeicher.
- Balanced Banana soll die Aufgaben der Benutzer auf die Arbeiter verteilen.
- Vom Benutzer in Auftrag gegebene Aufgaben sollen automatisch gestartet werden.
- Balanced Banana sammelt Statistiken über Aufgaben. Die Statistiken sind von dem Auftraggeber einsehbar.
- Öffentliche Statistiken sollen über eine Web-API mit HTTP-Anfragen bereitgestellt werden.
- Die Anzahl der Arbeiter ist variabel.
- Der Benutzer kann eine Priorität für jede Aufgabe angeben.
- Der Benutzer soll benachrichtigt werden, sobald eine von ihm in Auftrag gegebene Aufgabe erledigt ist oder fehlschlägt.
- Balanced Banana verfügt über eine ausführliche Dokumentation und Bedienungsanleitung.

2.2 Eigene Ergänzungen

- Der Benutzer kann eine von ihm in Auftrag gegebene Aufgabe pausieren.
- Balanced Banana kann die verbleibende Restlaufzeit für die nicht beendeten Aufgaben vorhersagen.
- Die Rückmeldung nach Beendigung einer Aufgabe kann in verschiedenen Sprachen erfolgen.
- Benutzer können den Status von Aufgaben, die nicht von ihnen in Auftrag gegeben wurden, einsehen und sich über Änderungen benachrichtigen lassen.
- Der Benutzer kann einen verzögerten Start einer Aufgabe anfordern.

2.3 Abgrenzungskriterien

- Balanced Banana führt die in Auftrag gegebenen Aufgaben nicht persönlich aus, sondern übergibt die Aufgabe an einen Arbeiter.
- Balanced Banana ignoriert die Ausgabe der Aufgaben. Balanced Banana ist ausschließlich für die Verteilung der Aufgaben verantwortlich.
- Balanced Banana ignoriert Fehler der Aufgaben. Eine Fehlerbehandlung soll von den Aufgaben oder dem Auftraggeber erfolgen.
- Die erhobenen Statistiken werden nicht ausgewertet. Die Statistiken werden zur Weiterverwendung durch die Benutzer erhoben.
- Balanced Banana garantiert nicht, dass getrennte Aufgaben miteinander kommunizieren können.

3 Szenarien

3.1 Einreihen einer Aufgabe in Warteschlange

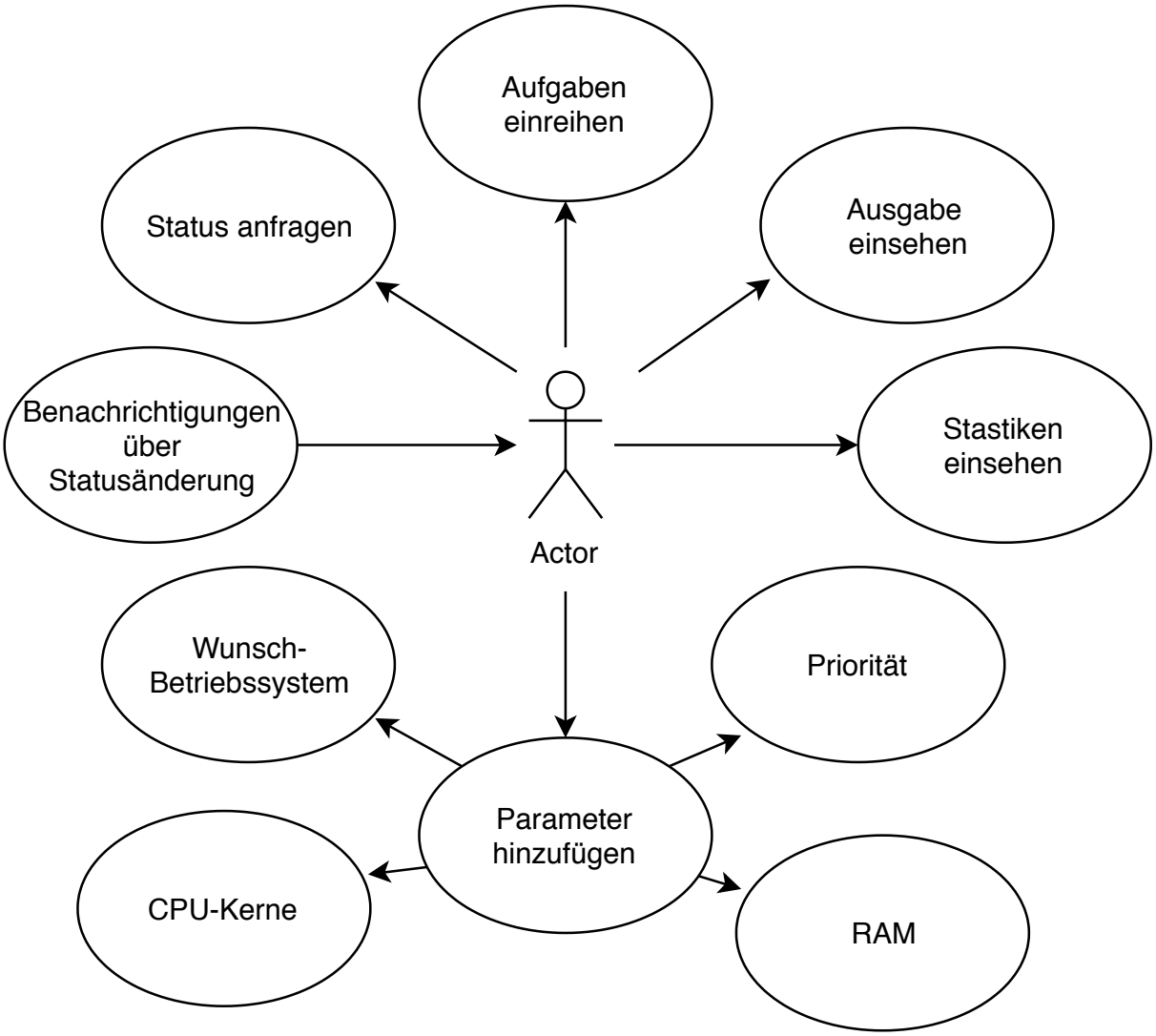
Tom möchte eine Katzensimulation simulieren lassen. Tom startet auf seinem PC den BalancedBanana-Client. Dieser verbindet sich im Hintergrund mit dem Server und authentifiziert Tom. Nun reiht Tom seine Simulation auf der Befehlszeile mit normaler Priorität ein. Tom sieht, dass alle Arbeiter ausgelastet sind und dass er an zweiter Stelle in der Warteschlange steht. Nach zwei Stunden sieht Tom erneut nach und stellt fest, dass er nun an vorderer Stelle in der Warteschlange steht. Nach drei weiteren Stunden sieht er, dass seine Aufgabe aus der Warteschlange verschwunden ist und er eine E-Mail erhalten hat. In dieser steht, dass seine Aufgabe in 43 Minuten abgeschlossen wurde er erhält einen Link mit dem er die Ausgabe seiner Aufgabe einsehen kann.

3.2 Aufwerten der Priorität

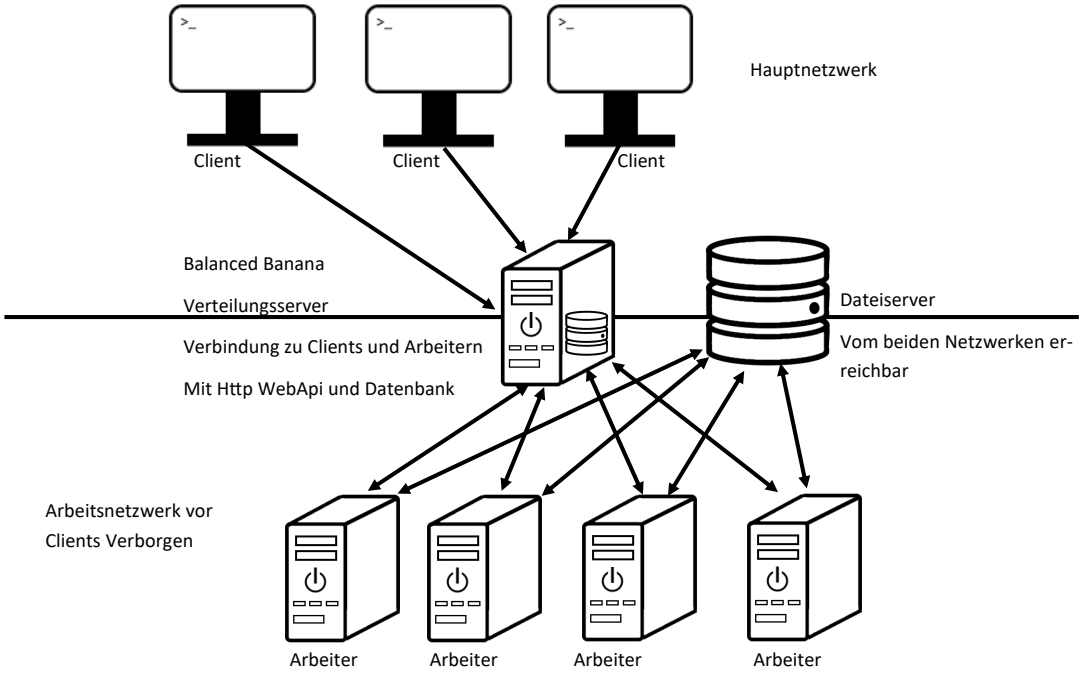
Peter möchte eine wichtige Bauteilsimulation rechnen lassen. Dazu startet Peter den BalancedBanana-Client und reiht seine Aufgabe mit normaler Priorität ein. Nachdem Peter nach 3 Stunden keine Rückmeldung bekommen hat, prüft er die Warteschlange und sieht das Tom ein Duzent Katzensimulationen mit hoher Priorität eingereiht hat. Genervt beendet Peter seinen PC und hofft das seine Simulation bis morgen abgeschlossen ist. Am nächsten morgen bemerkt Peter das Tom ein weiteres Duzent Katzensimulationen mit erneut hoher Priorität eingereiht hat. 24 Stunden nach Einreihung seiner Aufgabe wird nun die Priorität dieser automatisch erhöht und hat nun ebenfalls hohe Priorität. Peter bemerkt das seine Aufgabe nun in der Warteschlange langsam nach vorne rückt und schließlich gestartet wird. Nach einiger Zeit erhält Peter eine E-Mail mit dem Ergebnis seiner Simulation.

4 Systemmodelle

Die Grafik listet einige der Funktionen auf, die einem Benutzer zur Verfügung stehen.



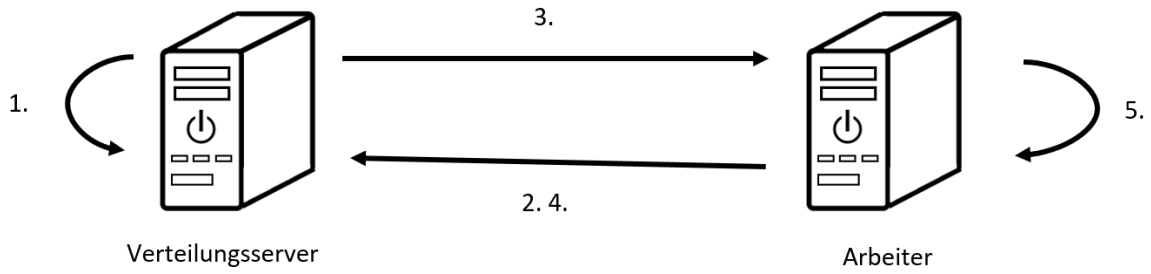
Den Kern des Systems stellt der Server dar. Er ist in der Lage über ein von Nutzern benutzbares Netzwerk Anfragen von den einzelnen Clients entgegenzunehmen und sie zu verwalten. Außerdem enthält er eine Datenbank, in der sämtliche Statistiken gespeichert werden, sowie einen HTTP Dienst, der Anfragen auf Statistik beantwortet und einen SMTP-Dienst, der die Benutzer benachrichtigt. An einer weiteren Schnittstelle ist er an ein von Außen nicht sichtbares Netzwerk angeschlossen, in dem sich die einzelnen Arbeiter befinden. Der Server kann Anfragen direkt an die sie in diesem Netz verteilen, ohne dass die Arbeiter von außerhalb des versteckten Netzes gesehen werden können. Außerdem gibt es ein geteiltes Netzwerk-Dateisystem, das von allen Rechnern in beiden Netzwerken, dem öffentlichen Netz und dem versteckten Arbeiternetz, erreichbar ist. Hier können Daten und Programme gespeichert werden, die für die erstellten Aufgaben benötigt werden.



4.1 Ablaufdiagramme

4.1.1 Systeminitialisierung

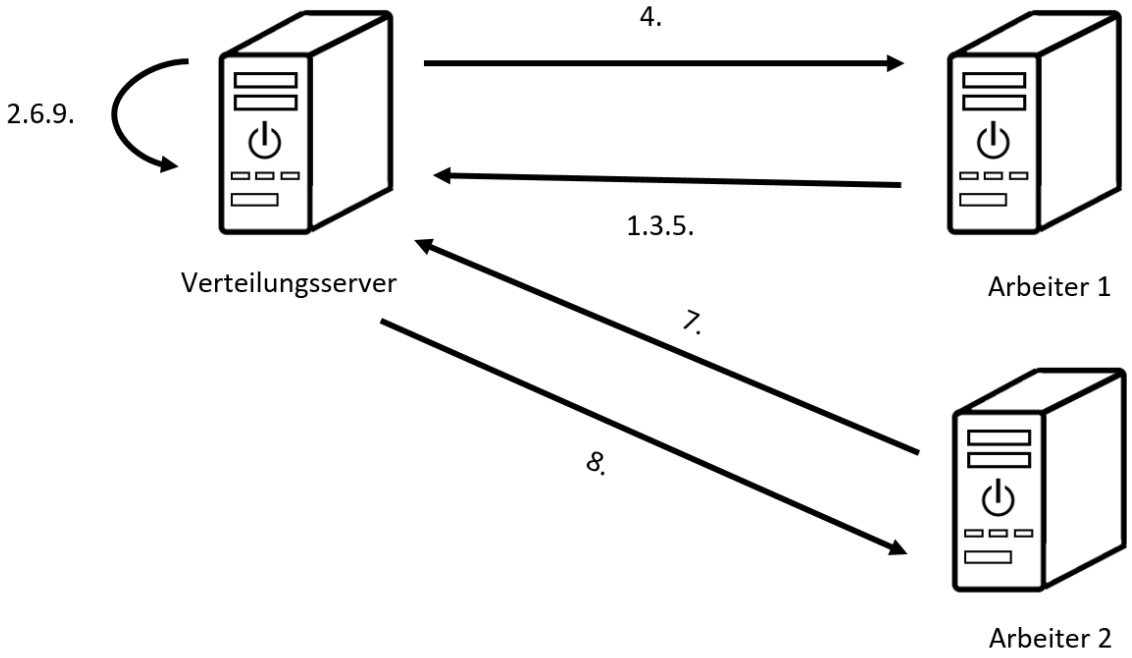
Dieses Ablaufdiagramm soll grob den Ablauf der Systeminitialisierung skizzieren, bei der aus einzelnen Rechnern das ganze System gebaut wird.



1. Der Server fängt an, auf Anfragen von Nutzern und Arbeitern zu hören. Falls die Warteschlange beim letzten System Stopp nicht leer war, lädt er außerdem alle noch nicht bearbeiteten Aufgaben in seine Warteschlange.
2. Ein Arbeiter schickt eine Anfrage an alle Rechner im Netzwerk, ob es sich bei ihnen um einen Server handelt.
3. Der Server bestätigt dem Arbeiter, dass er ein Server ist.
4. Der Arbeiter teilt dem Server mit, dass er bereitsteht um bei jenem eingereichte Aufgaben zu bearbeiten.
5. Der Arbeiter wartet auf Aufgaben vom Server.

4.1.2 Systemabbau

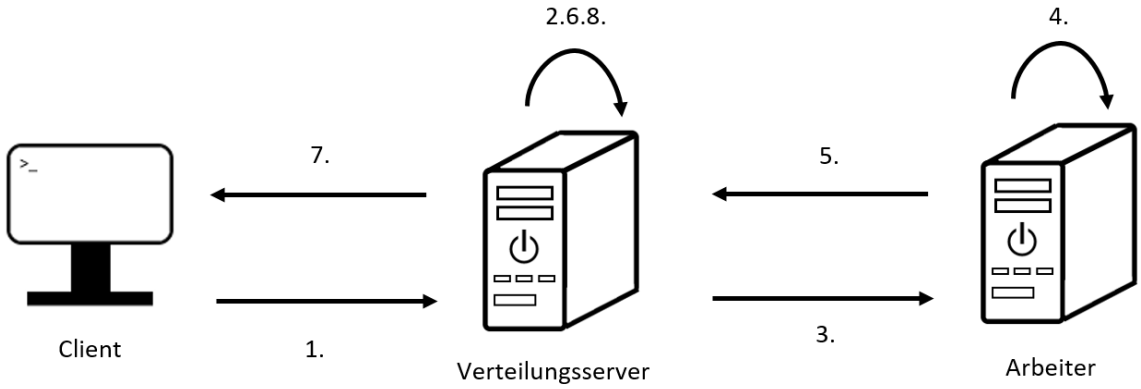
Im Folgenden soll der Ablauf zur Außerbetriebnahme des Systems für beispielsweise Wartungsarbeiten skizziert werden.



Manuelle Abmeldung eines einzelnen Arbeiters:
1. Der Arbeiter1 soll vom System abgemeldet werden und meldet sich deshalb beim Server ab, rechnet aber seine aktuelle Aufgabe noch zu ende.
2. Der Server plant den Arbeiter1 aus.
3. Der Arbeiter1 meldet den Abschluss der Aufgabe.
4. Der Server benachrichtigt den Arbeiter1, dass er befreit ist.
5. Der Arbeiter1 bestätigt dem Server, dass er die Befreiung erhalten hat.
Herunterfahren des gesamten Systems:
6. Der Server wartet, bis alle Arbeiter mit ihren aktuellen Aufgaben fertig sind, gibt sie dann frei (siehe 4. bis 5.) und speichert schon mal alle sich noch in der Warteschlange befindenden Aufgaben ab.
7. Der letzte Arbeiter (Arbeiter2) meldet den Abschluss seiner Aufgabe. 8. Der Server verarbeitet diese Meldung normal und gibt auch ihn frei. 9. Der Server beendet seinen Dienst.

4.2 Aufgabenverarbeitung

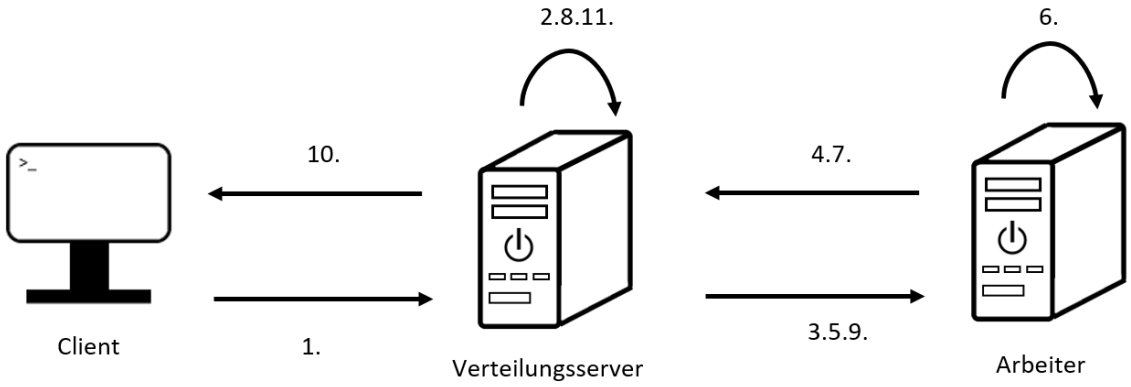
Dieses Diagramm beschreibt den groben Verlauf der Bearbeitung einer Aufgabe von der Erstellung bis zur Benachrichtigung beim Abschluss.



1. Der Benutzer erstellt eine Aufgabe und sendet sie an den Server. 2. Der Server plant die Aufgabe ein und wartet, bis ein Arbeiter genug Ressourcen zur Verfügung stellt, damit die Aufgabe bearbeitet werden kann.
3. Der Server sendet die Aufgabe an den Arbeiter.
4. Der Arbeiter bearbeitet die Aufgabe.
5. Der Arbeiter meldet den Abschluss der Aufgabe beim Server.
6. Der Server speichert Statistiken über die Aufgabe in der Datenbank ab.
7. Der Server meldet den Abschluss der Aufgabe an den Benutzer, sofern er noch im System angemeldet ist und auf eine Antwort wartet.
8. Der Server verschickt eine E-Mail an die bei der Erstellung der Aufgabe spezifizierte Adresse, die den Benutzer auf den Abschluss seiner Aufgabe hinweist.

4.2.1 Notfall Aufgabenverarbeitung

Dieses Diagramm veranschaulicht den Verlauf einer Notfall Verteilung. Eine Notfall Verteilung bezeichnet eine Aufgabe, die schnellstmöglich bearbeitet werden muss.



1. Der Client sendet eine Notfall-Anfrage an den Server.
2. Der Server sucht sich einen geeigneten Arbeiter.
3. Der Server weist den Arbeiter dazu an, seine Arbeit zu unterbrechen.
4. Der Arbeiter meldet, dass er seine Arbeit unterbrochen hat.
5. Der Server übermittelt die Notfall Aufgabe an den Arbeiter.
6. Der Arbeiter bearbeitet die Notfall Aufgabe.
7. Der Arbeiter benachrichtigt den Server über den Abschluss des Notfalls.
8. Der Server speichert Statistiken und Ergebnis der Notfall Aufgabe.
9. Der Server weist den Arbeiter an, seine unterbrochene Arbeit fortzusetzen.
10. Der Server benachrichtigt den Client über den Abschluss des Notfalls.
11. Der Server löst das Versenden der Benachrichtigungs E-Mail aus.

5 Produkteinsatz

- **Zielgruppe**
  Balanced Banana ist für die Verwendung durch die Mitarbeiter am Chair for Embedded Systems (CES) des Karlsruher Institut für Technologie (KIT) gedacht.
  Den Mitarbeitern soll die Benutzung der Arbeiter erleichtert werden, indem die Verteilung der Aufgaben automatisch erfolgt. Somit soll garantiert werden, dass die Aufgaben sich gegenseitig möglichst wenig stören.
  Durch die Vermeidung von Situationen, in denen mehrere Personen Aufgaben auf dem selben Arbeiter ausführen, während einer oder mehrere andere Arbeiter unbeschäftigt sind, soll die Leistungsfähigkeit sowie allgemeine Zufriedenheit und Harmonie gesteigert werden.

- **Verwendungszweck**
  Balanced Banana dient dazu, Simulationen und andere rechenintensive Aufgaben auf eigens zu diesem Zweck gedachte Rechner (Arbeiter) zu verteilen.
  Oftmals fallen Aufgaben an, die von einem Computer berechnet werden können und sollen. Nicht immer ist der eigene Rechner jedoch dieser Aufgabe gewachsen. Damit sich Benutzer und Aufgabe also nicht um Rechenzeit streiten, werden die anstehenden Aufgaben auf einen Pool von Arbeitern verteilt, die speziell für diesen Zweck zur Verfügung stehen. Das Verteilen wird von Balanced Banana übernommen.

- **Produktaufbau**
  Balanced Banana ist den Aufgabengebieten entsprechend in drei Teile aufgeteilt:
  1. Der Client (Benutzer): Verantwortlich für die Interaktion des Benutzers mit dem Produkt auf der Befehlszeile.
  2. Der Server: Verantwortlich für das effiziente Verteilen der Aufgaben auf dem Arbeiterpool. Agiert als Mittelmann zwischen Benutzer und Arbeiter.
  3. Der Arbeiter (Rechenknoten): Verantwortlich für Ausführung, Pausieren und Abbruch der Aufgaben auf den Arbeitern (spezielle Rechner für die Aufgabenverarbeitung).

6 Funktionale Anforderungen

6.1 Übersicht der Funktionalen Anforderungen

FA1	Client verbindet sich beim Starten mit dem Server.
FA2	Benutzer authentifiziert sich über den Client gegenüber dem Server.
FA3	Benutzer kann eine Aufgabe über den Client einreihen.
FA4	Benutzer kann Parameter übergeben.
FA41	Client speichert voreingestellte Parameter in Konfigurationsdatei.
FA42	Benutzer kann Parameter über Konfigurationsdatei übergeben.
FA43	Benutzer kann Priorität einer Aufgabe festlegen.
FA44	Benutzer kann minimale und maximale Anzahl genutzer Kerne festlegen.
FA45	Benutzer kann maximal nutzbaren Arbeitsspeicher festlegen.
FA46	Benutzer kann das benutzte Betriebssystem festlegen.
FA47	Benutzer kann angeben, ob der Client blockieren soll, bis die Aufgabe beendet ist.
FA48	Benutzer übergibt Pfad zu der Aufgabe.
FA49	Falls der Benutzer keine Parameter übergibt, werden Standardwerte benutzt.
FA5	Benutzer kann den Status einer Aufgabe einsehen.
FA6	Benutzer bekommt Benachrichtigung über abgeschlossene Aufgabe.
FA7	Server erstellt regelmäßige Sicherungen von laufenden Aufgaben.
FA8	Benutzer kann die Ausgabe seiner Aufgabe anfordern.
FA9	Benutzer kann erhobene Statistiken abfragen.

6.2 Erläuterung der funktionalen Anforderungen

FA1 **Client verbindet sich beim Starten mit dem Server.**

Erklärung: Beim Starten des Client soll dieser sich automatisch mit dem Server verbinden.

Voraussetzung(en): Keine.

Nachbedingung(en):

Erfolg: Der Client hat sich mit dem Server verbunden.

Misserfolg: Der Client konnte sich nicht mit dem Server verbinden. Eine entsprechende Fehlermeldung wird ausgegeben.

FA2 **Benutzer authentifiziert sich über den Client gegenüber dem Server.**

Erklärung: Bevor der Benutzer Befehle ausführen kann, muss dieser sich authentifziert haben.

Voraussetzung(en): Der Client hat sich mit dem Server verbunden.

Nachbedingung(en):

Erfolg: Der Benutzer wurde authentifiziert.

Misserfolg: Der Benutzer konnte nicht authentifiziert werden. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Bevor der Benutzer eine Aufgabe einreihen kann (FA3) oder den Status seiner Aufgaben einsehen kann (FA6), muss sich dieser authentifizieren. Dies führt der Client automatisch durch. Falls notwendig, wird der Benutzer aufgefordert ein Passwort einzugeben.

FA3 **Benutzer kann eine Aufgabe über den Client einreihen.**

Erklärung: Benutzer kann eine Aufgabe mit Parametern zur Bearbeitung einreihen.

Voraussetzung(en): Der Benutzer hat sich gegenüber dem Server authentifiziert.

Nachbedingung(en):

Erfolg: Die Aufgabe wird in eine Warteschlange eingereiht. Der Benutzer bekommt eine Rückmeldung mit der Aufgaben ID.

Misserfolg: Die Aufgabe wird nicht eingereiht. Der Benutzer bekommt eine entsprechende Fehlermeldung.

Details: Der Benutzer kann über die Befehlszeile eine Aufgabe einreihen. Dabei kann er Parameter mit übergeben (FA4). Falls der Benutzer keine Parameter mit übergibt werden Standardparameter verwendet (FA49).

FA4 **Benutzer kann Parameter übergeben.**

Erklärung: Bei der Einreihung einer Aufgabe, können weiter Parameter übergeben werden.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Der Befehl wird mit den angegebenen Parametern eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Wenn eine Aufgabe eingereiht wird, kann der Benutzer noch weitere Parameter hinzufügen. Diese können in einer Konfigurationsdatei übergeben werden (FA42). Falls keine Parameter eingegeben wurden, werden Standardparameter verwendet (FA49). Können mehrere Rechner für eine Aufgabe verwendet werden, so wird zuerst nach minimalem Arbeitsspeicher und danach nach minimalen Kernen ausgesucht.

FA41 **Client speichert voreingestellte Parameter in Konfigurationsdatei.**

Erklärung: Standardparameter werden einer Konfigurationsdatei gespeichert. Diese können bearbeitet werden. Zur einfachen Einreihung können alle Parameter durch eine Konfigurationsdatei angegeben werden.

Vorraussetzungen: Keine.

FA42 **Benutzer kann Parameter über Konfigurationsdatei übergeben.**

Erklärung: Gibt der Benutzer Parameter bei der Einreihung einer Aufgabe nicht ein, so werden diese von der Konfigurationsdatei bezogen.

Vorraussetzungen: Eine Aufgabe soll eingereiht werden.

Nachbedingung:

Erfolg: Fehlende Parameter wurden aus der Konfigurationsdatei bezogen.

Misserfolg: Fehlende Parameter werden auf Standardwerte gesetzt.

Details: Werden Parameter bei der Einreihung einer Aufgabe nicht weiter spezifiziert, so werden diese aus der Konfigurationsdatei gelesen. Sind diese dort ebenfalls nicht spezifiziert, so werden diese auf Standardwerte gesetzt.

FA43 **Benutzer kann Priorität einer Aufgabe festlegen.**

Erklärung: Benutzer kann seine Aufgabe mit einer Priorität versehen.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird mit der gewünschten Priorität in die Warteschlange eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann einer eingereihten Aufgabe eine Priorität geben. Diese bestimmt welche Position die Aufgabe in der Warteschlange bekommt. Es gibt vier Prioritäten: Low, Normal, High, Emergency (von niedrig bis hoch sortiert). Eine höhere Priorität bewirkt ein Einreihen weiter vorne in der Warteschlange. Bei zwei Aufgaben mit gleicher Priorität entscheidet der Zeitpunkt der Einreihung.

FA44 **Benutzer kann minimale und maximale Anzahl genutzer Kerne festlegen.**

Erklärung: Benutzer kann eine minimale und maximale Anzahl nutzbarer Kerne als Parameter angeben.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird mit der gewünschten minimalen und maximalen Anzahl Kerne als Parameter in die Warteschlange eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann einer eingereihten Aufgabe eine minimale und maximale Anzahl nutzbarer Kerne zuweisen. Die Aufgabe wird dann mindestens die spezifizierte Angabe Kerne nutzen können und maximal die spezifizierte Anzahl Kerne nutzen können.

FA45 **Benutzer kann minimal und maximal nutzbaren RAM festlegen.**

Erklärung: Erlaubt es dem Benutzer eine minimale bzw. maximale Menge Arbeitsspeicher als Parameter anzugeben.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird mit der gewünschten minimalen und maximalen Größe des nutzbaren Arbeitsspeichers als Parameter in die Warteschlange eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann einer eingereihten Aufgabe eine minimale und maximale Größe nutzbaren RAMs zuweisen. Die Aufgabe wird dann mindestens die minimale Größe und höchstens die maximale Größe nutzen können.

FA46 **Benutzer kann das benutzte Betriebssystem festlegen.**

Erklärung: Erlaubt es dem Benutzer das Betriebssystem, auf dem die Aufgabe ausgeführt werden soll, als Parameter auszuwählen.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird mit dem gewünschten Betriebssystem als Parameter eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann einer eingereihten Aufgabe ein gewünschtes Betriebssystem als Parameter zuweisen. Die Aufgabe wird dann auf dem gewünschten Betriebssystem ausgeführt.

FA47 **Benutzer kann angeben, ob der Client blockieren soll, bis die Aufgabe beendet ist.**

Erklärung: Erlaubt es dem Benutzer anzugeben, ob der Client keine weiteren Befehle annehmen soll, bis die einreihende Aufgabe beendet ist.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird eingereiht. Der Benutzer kann keine weiteren Befehle über den Client eingeben, bis die eingereihte Aufgabe beendet ist.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann zu einer eingereihten Aufgabe spezifizieren, ob der Client Eingaben blockieren soll, bis die eingereihte Aufgabe beendet wurde. Dies ist nützlich für Skripte für z.B Aufgaben, die abhängig von dem Ergebnis von anderen Aufgaben sind.

FA48 **Benutzer übergibt Pfad zu der Aufgabe.**

Erklärung: Der Benutzer ist aufgefordert einen Dateipfad als Parameter an eine Einreihung anzuhängen, der zu der Aufgabe zeigt.

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die angegebene Aufgabe wird eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer wird aufgefordert einen Dateipfad als Parameter an einen Einreihungsbefehl anzuhängen. Hierfür existiert kein Standardparameter. Die Ausführbarkeit der Aufgabe wird bis zum Aufgabenstart nicht überprüft.

FA49 **Falls der Benutzer keine Parameter übergibt, werden Standardwerte benutzt.**

Erklärung: Sollte der Benutzer keine Parameter über die Befehlszeile übergeben oder nicht in einer Konfigurationsdatei spezifiziert haben, werden für bestimmte Parameter Standardwerte ausgewählt

Voraussetzung(en): Eine Aufgabe soll eingereiht werden.

Nachbedingung(en):

Erfolg: Die Aufgabe wird mit Standardparametern eingereiht.

Misserfolg: Die Aufgabe wird nicht eingereiht. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Sollten einzelne Parameter in der Befehlseingabe fehlen, so sucht sich der Client diese fehlende Parameter aus der Konfigurationsdatei. Sollten diese dort ebenfalls nicht spezifiziert werden, so werden für einige Standardparameter verwendet. Diese sind für alle Clients die selben. Die Standardwerte sind änderbar.

Standardparameter:
Priorität: Normal
Minimale Kerne: 1
Maximale Kerne: 12
Minimal nutzbarer RAM: 1 GB
Maximal nutzbarer RAM: 32 GB
Betriebssystem: Egal
blockierend: Nein
Pfad: Kein Standardparameter (Gibt Fehlermeldung aus, falls fehlend)

FA5 **Benutzer kann den Status einer Aufgabe einsehen.**

Erklärung: Der Benutzer kann den Status seiner Aufgabe anhand einer ID von dem Client aus einsehen. Hierzu gehören Wartezeit, Position in der Warteschlange, Priorität und Zeit seitdem die Aufgabe auf einem Arbeiter gestartet wurde.

Voraussetzung(en): Eine Aufgabe wurde vom Benutzer eingereiht.

Nachbedingung(en):

Erfolg: Eine entsprechende Statusmeldung wird angezeigt

Misserfolg: Keine Statusmeldung wird angezeigt. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Ein Benutzer kann den Status einer seiner Aufgaben anhand der ID vom Client aus abfragen. Hierbei bekommt er Auskunft, wie lange seine Aufgabe schon wartet, an welcher Position in der Warteschlange sie sich befindet, welche Priorität die Aufgabe besitzt und, falls die Aufgabe bereits gestartet wurde, wie lange sie bereits läuft.

FA6 **Benutzer bekommt Benachrichtigung über abgeschlossene Aufgabe.**

Erklärung: Nach Abschluss einer Aufgabe soll der Benutzer eine Benachrichtigung in Form einer E-Mail erhalten.

Voraussetzung(en): Eine Aufgabe wurde abgeschlossen.

Nachbedingung(en):

Erfolg: Der Benutzer bekommt eine E-Mail als Benachrichtigung der abgeschlossenen Aufgabe.

Misserfolg: Der Benutzer bekommt keine E-Mail als Benachrichtigung.

Details: Sollte der Benutzer eine E-Mail als Parameter zu einer Einreihung einer Aufgabe mitgegeben haben, so bekommt dieser eine Benachrichtigung über den Abschluss dieser Aufgabe. Sollte diese E-Mail Adresse nicht existieren, so kann der Benutzer den Status seiner Aufgabe auch mit der Aufgaben ID auslesen.

FA7 **Server erstellt regelmäßige Sicherungen von laufenden Aufgaben.**

Erklärung: Der Server erstellt eigenständig in regelmäßigen Intervallen Sicherungen von laufenden Aufgaben.

Voraussetzung(en): Mindestens eine Aufgabe wird ausgeführt.

Nachbedingung(en):

Erfolg: Es werden Sicherungen erstellt.

Misserfolg: Es werden keine Sicherungen erstellt. Eine entsprechende Fehlermeldung wird aufgezeichnet.

Details: In regelmäßigen Abständen (Standard: 1 Stunde) erstellt der Server eine Sicherung von allen laufenden Aufgaben. Diese können zu einem späteren Zeitpunkt gestartet werden, sollte das System die Aufgaben nicht beendet haben. Der Server speichert alle Sicherungen der letzten Woche. Die Sicherungsintervalle können angepasst werden.

FA8 **Benutzer kann die Ausgabe seiner Aufgabe anfordern.**

Erklärung: Der Benutzer kann die Ausgabe einer seiner Aufgaben anhand der ID anfordern.

Voraussetzung(en): Keine.

Nachbedingung(en):

Erfolg: Dem Benutzer werden die letzten 200 Zeilen der Ausgabe seiner Aufgabe angezeigt.

Misserfolg: Dem Benutzer wird keine Ausgabe angezeigt. Eine entsprechende Fehlermeldung wird ausgegeben.

Details: Der Benutzer kann anhand einer ID die Ausgabe seiner Aufgabe anfordern. Die letzten 200 Zeilen werden auf der Konsole ausgegeben.

FA9 **Benutzer kann erhobene Statistiken abfragen.**

Erklärung: Zu den ausgeführten Aufgaben werden Statistiken erhoben.

Voraussetzung(en): Keine.

Nachbedingung(en):

Erfolg: Der Benutzer erhält angefragte Auszüge der gespeicherten Statistiken.

Misserfolg: Der Benutzer erhält keine Statistiken. Eine entsprechende Fehlermeldung wird gesendet.

Details: Es werden Statistiken zu den individuellen Aufgaben, sowie zu dem Gesamtsystem erhoben. Diese können über einen HTTPS-Server angefragt werden.

6.3 Übersicht der optionalen Anforderungen

OFA1 Benutzer kann eine geschätzte Restzeit einer Aufgabe sehen.
OFA2 Server stoppt Aufgaben die zu lange dauern.
OFA3 Benutzer kann eine manuelle Stoppung seiner Aufgabe anfordern.
OFA4 Benutzer kann eine manuelle Sicherung seiner Aufgabe anfordern.
OFA5 Benutzer kann angeben ob die Aufgabe pausierbar ist.

7 Produktdaten

PD1 **Auftraggeber Kontaktinformation**

Erklärung Hält Kontaktinformation (E-Mail Adresse) des Auftraggebers.

Details Um dem Auftraggeber Rückmeldung über abgeschlossene Aufgaben geben zu können, muss eine Kontaktinformation hinterlegt sein.
Es ist vorgesehen, eine E-Mail Adresse als Kontaktinformation anzunehmen. Diese wird beim Einreichen der Aufgabe von dem Auftraggeber angegeben.

PD2 **NutzerID**

Erklärung Identifiziert einen Benutzer

Details Das Programm soll die Identität eines Benutzers sicherstellen können. Zu diesem Zweck ist eine ID notwendig, die dem Benutzer ausgestellt wird. Die ID kann dazu verwendet werden, um Berechtigung für diverse Anfragen zu prüfen. So soll zum Beispiel nur der Auftraggeber dazu in der Lage sein, den Status der Aufgabe einzusehen.

PD21 **Zuweisung Nutzer zu Aufgaben**

Erklärung Speichert die in Auftrag gegebenen Aufgaben eines Auftraggebers

Details Um dem Benutzer zu ermöglichen, den Status seiner Aufgaben zu verfolgen, muss eine Zuweisung von Nutzer zu Aufgaben möglich sein.

PD3 **Arbeiter Liste**

Erklärung Eine Liste aller verfügbaren Arbeiter.

Details Enthält alle verfügbaren Arbeiter.
Somit kann der Server die Aufgaben auf alle Arbeiter verteilen.

PD4 **Last**

Erklärung Tatsächliche Auslastung der Arbeiter einzeln und gemeinsam.

Details Die Auslastung der Hardware (CPU, RAM) ist für jeden Arbeiter individuell bekannt.
Das Mittel der Hardwareauslastung aller Arbeiter (CPU, RAM) ist bekannt.

PD5 **Aufgaben**

Erklärung Liste aller ausstehenden Aufgaben

Details Es ist bekannt, welche Aufgaben derzeit ausstehen (noch nicht beendet sind). Weiter ist bekannt welche dieser Aufgaben derzeit ausgeführt werden (aktiv) und welche derzeit warten (passiv).

PD51 **Zuweisung Aufgabe zu Arbeiter**

Erklärung Aufgaben laufen entweder auf keinem (passiv) oder auf genau einem (aktiv) Arbeiter

Details Um genauer Informationen über den Status einer Aufgabe zu erhalten, muss bekannt sein, welche Aufgabe auf welchem Arbeiter läuft.
Ist eine Aufgabe passiv, ist ihr kein Arbeiter zugewiesen.
Ist eine Aufgabe aktiv, ist ihr immer genau ein Arbeiter zugewiesen.

PD52 **Prioritäten der Aufgaben**

Erklärung Summe aller Aufgaben einer bestimmten Priorität

Details Summiert für jede Priorität die Anzahl der derzeit aktiven und passiven Aufgaben auf, die mit dieser Priorität versehen sind.

PD53 **BefehlsParameter**

Erklärung Die Werte der BefehlsParameter

Details Für jede Aufgabe ist der Wert der Parameter wie er auf der Befehlszeile spezifiziert wurde, sowie der tatsächlich verwendete Wert bekannt.

PD54 **Ausgabe und Ergebnis**

Erklärung Ausgaben der Aufgabe und Rückgabewert der Aufgabe

Details Die Ausgabe der Aufgabe erfolgt auf der Befehlszeile. Sie enthält von der Aufgabe generierte Informationen für den Benutzer.
Das Ergebnis der Aufgabe ist der Rückgabewert. Er liefert Auskunft über den Erfolg der Ausführung.

PD55 **Arbeitszeiten**

Erklärung Angaben zu den Zeiten die eine Aufgabe im System verbracht hat.

Details Die Informationen über die Arbeitszeit einer Aufgabe sind aufgeteilt in:

1. aktive Zeit: Die Zeit, in der die Aufgabe tatsächlich ausgeführt wurde.
2. passive Zeit: Die Zeit in der die Aufgabe in der Warteschlange verbracht hat.
3. Gesamtzeit: Die Zeit die zwischen Auftragseingang und Abschicken der Abschlussbenachrichtigung vergangen ist.

8 Nichtfunktionale Anforderungen

8.1 Übersicht der Anforderungen

NFA1 Bei Abschluss einer Aufgabe soll die Rückmeldung innerhalb von 60 Minuten erfolgen.
NFA2 Ein Benutzer darf nur auf eigene Dateien zugreifen.
NFA3 Statistiken abgeschlossener Aufgaben sollen nicht änderbar sein.
NFA4 Passwörter werden nicht als Klartext gespeichert werden.
NFA5 Der Benutzer verbindet sich mit dem Server über eine sichere Verbindung.

8.2 Erläuterung der nichtfunktionalen Anforderungen

NFA1 **Rückmeldung erfolgt innerhalb von 60 Minuten**
Erklärung: Bei Abschluss einer Aufgabe soll die Rückmeldung innerhalb von 60 Minuten erfolgen. Dies ist auch der Fall, wenn eine Aufgabe fehlschlägt. Die Rückmeldung ist per E-Mail zu erfolgen.

NFA2 **Ein Benutzer darf nur auf eigene Dateien zugreifen**
Erklärung: Um die Sicherheit der Daten jedes Benutzers zu gewährleisten, ist es normalen Benutzern nicht gestattet, auf die Daten eines anderen Benutzers zuzugreifen.

NFA3 **Statistiken abgeschlossener Aufgaben sollen nicht änderbar sein**
Erklärung: Statistiken sollen nicht veränderbar sein, um die Zuverlässigkeit und Genauigkeit der Daten zu gewährleisten.

NFA4 **Passwörter werden nicht als Klartext gespeichert werden**
Erklärung: Um die Sicherheit privater Benutzerdaten zu gewährleisten, sind Passwörter zu verschlüsseln.

NFA5 **Benutzer verbindet sich mit dem Server über eine sichere Verbindung**
Erklärung: Um die Sicherheit der Benutzerdaten zu gewährleisten, müssen sich die Benutzer über eine sichere Verbindung mit dem Server verbinden.

9 Produktumgebung

- **Client (Nutzer) Umgebung**
  Das Benutzer Produkt ist für den Einsatz auf einem Rechner bestimmt, der folgenden Anforderungen genügt:
  Betriebssystem: Linux basiertes Betriebssystem
  CPU: 1 GHz oder schneller
  Arbeitsspeicher: Mindestens 1 GB verfügbarer Arbeitsspeicher

- **Scheduler (Server) Umgebung**
  Das Server Produkt ist für den Einsatz auf einem Rechner bestimmt, der das Benutzer Netzwerk mit dem Arbeiter Netzwerk verbindet und folgenden Anforderungen genügt:
  Betriebssystem: CentOS 7 oder Ubuntu 18.04
  CPU: 1 GHz oder schneller
  Arbeitsspeicher: Mindestens 4 GB verfügbarer Arbeitsspeicher

- **Arbeiter (Arbeiter) Umgebung**
  Das Arbeiter Produkt ist für den Einsatz auf einem Rechner bestimmt, der folgenden Anforderungen genügt:
  Betriebssystem: CentOS 7 oder Ubuntu 18.04
  CPU: 1 GHz oder schneller
  Arbeitsspeicher: Mindestens 8 GB verfügbarer Arbeitsspeicher,

10 Benutzer Oberfläche

10.1 Beispiel E-Mails

10.1.1 Aufgabe gestartet

Dies ist ein Beispiel für eine E-Mail-Benachrichtigung, die ein Benutzer erhält, wenn eine Aufgabe ausgeführt wird, und nicht, wenn sie vom Benutzer in die Warteschlange aufgenommen wird.

Balanced Banana - Status change

balancedbanana <balancedbanana@protonmail.com>
to me

Dear Messrs,

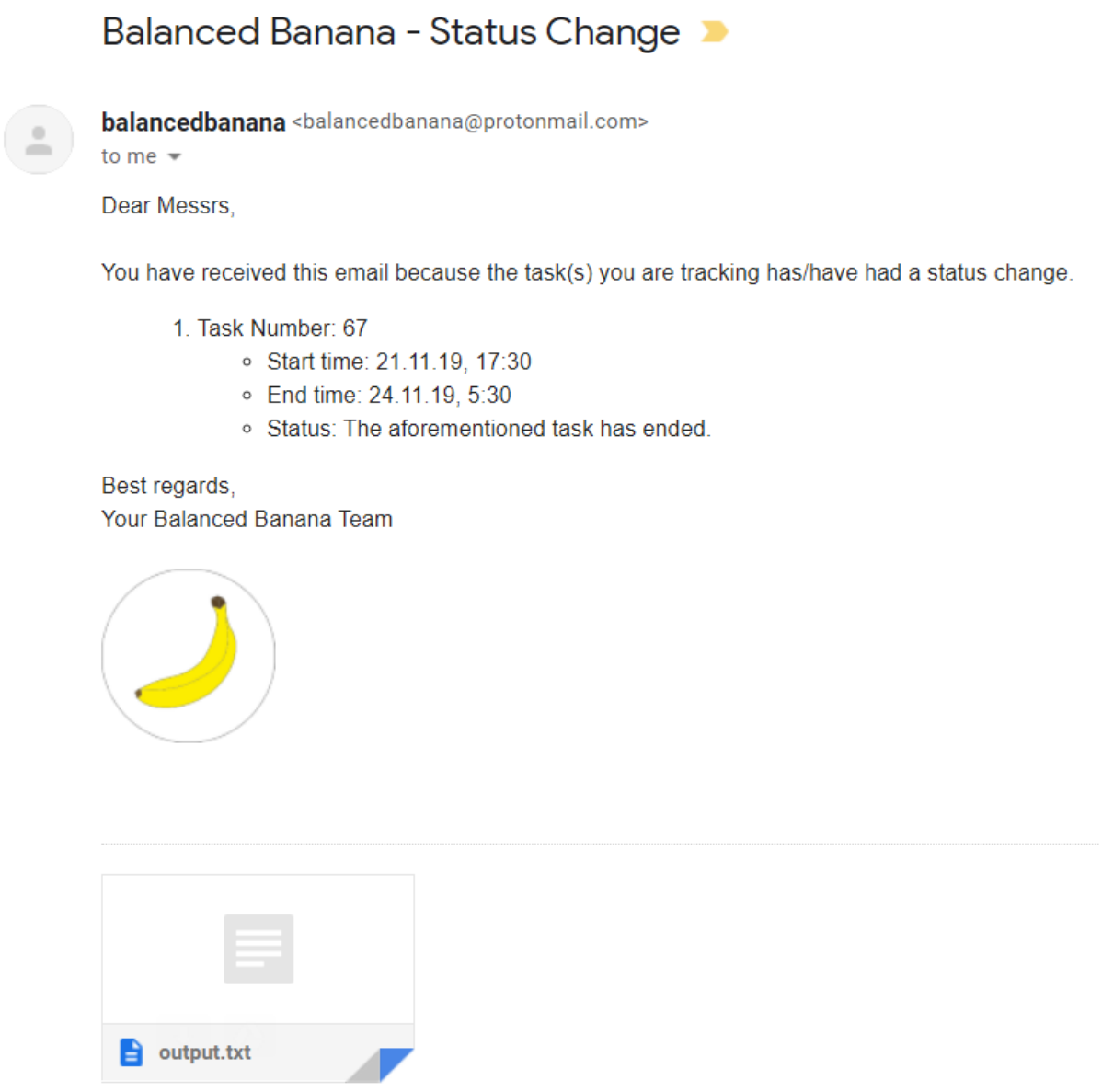
You have received this email because the task(s) you are tracking has/have had a status change.

1. Task Number: 1
    - Start time: 24.11.19, 17:30
    - End time: N/A
    - Status: The aforementioned task has begun.

Best regards,
Your Balanced Banana Team

10.1.2 Aufgabe beendet

Dies ist ein Beispiel für eine E-Mail-Benachrichtigung, die ein Benutzer erhält, wenn eine Aufgabe abgeschlossen ist. Eine Aufgabe gilt als abgeschlossen, wenn sie erfolgreich ausgeführt wurde oder fehlgeschlagen ist. Die E-Mail-Benachrichtigung zeigt diesen Unterschied nicht an, es liegt an dem Benutzer, das Programm zu überprüfen, um zu sehen, ob die Aufgabe fehlgeschlagen ist oder nicht. Die Ausgabe des Programms wird in einer .txt-Datei gespeichert und in der E-Mail angehängt.



10.2 Befehlszeile

10.2.1 HauptServer starten

Startet das Programm im Daemon Modus. Optional kann die Abhöhrende Internet Protokoll Adresse (v4 oder v6) und den Port (Dezimale 16bit Ganzzahl) im (lokalen) Netzwerk und der Http Web-API angeben werden.

```
bbs [<--server|-s> <ipaddr>] [<--serverport|-sp> <port>] [<--webapi}|-w>
<ipaddr>] [<--webapi-port|-wp> <port>]
```

10.2.2 Arbeiter starten

Startet das Programm im Daemon Modus. Optional kann die Internet Protokoll Adresse (v4 oder v6) bzw. der DNS Namen des HauptServers und den Port (Dezimale 16bit Ganzzahl) im (lokalen) Netzwerk angeben werden.

```
bbd [<--server|-s> <ipaddr>] [<--serverport|-sp> <port>]
```

10.2.3 Aufgabe erstellen

Sendet die angegebene Aufgabe an den Server und gibt die Aufgaben ID an der Standard Ausgabe aus (Dezimal).

```
bbc <--run|-r> [--block|-b] [<--priority|-p> <priority>] [<--max-cpu-count|-Mc>
<Integer>] [<--min-cpu-count|-mc> <Integer>] [<--max-ram|-Mr> <Integer>]
[<--min-ram|-mr> <Integer>] [<--email|-em> <email>] [<--image|-im>
<name>] [<program> [args]+]
```

Falls `--block` block angegeben wurde, wird statt der Aufgaben ID die Ausgabe der Aufgabe ausgegeben. Außer man beendet es forzeitig mit `CTRL + C`, dann wird die Aufgaben ID direkt danach ausgeben.

10.2.4 Status der Aufgabe anfordern

Gibt den Status der angegebenen Aufgabe aus.

```
bbc <--status|-s> <id>
```

`<id>` Aufgaben ID die beim erstellen der Aufgabe ausgegeben wurde.

10.2.5 Ausgabe der Aufgabe ansehen

Gibt nur die letzten Zeilen der Standardausgabe der angegebenen Aufgabe aus, bis man mit `CTRL + C` abbricht oder die Aufgabe sich beendet.

```
bbc <--tail|-t> <id>
```

`<id>` Aufgaben ID die beim erstellen der Aufgabe ausgegeben wurde.

10.2.6 Aufgaben sichern

Erstellt einen Schnapschuss der Aufgabe, der später bei bedarf wiederhergestellt werden kann. Gibt die Sicherungs ID an der Standard Ausgabe aus (Dezimal).

`bbc <--backup|-b> <id>`

`<id>` Aufgaben ID die beim erstellen der Aufgabe ausgegeben wurde.

10.2.7 Aufgaben fortsetzen

Setzt die angehaltene Aufgabe fort.

`bbc <--continue|-c> <id>`

10.2.8 Aufgaben wiederherstellen

`bbc <--restore|-r> <id> <backupid>`

Sicherungs ID die beim erstellen der Sicherung ausgegeben wurde.

10.2.9 Aufgaben pausieren

Pausiert die Aufgabe mit der angegeben ID Nummer.

`bbc <--pause|-p> <id>`

10.2.10 Aufgaben beenden

Beendet die Aufgabe mit der angegeben ID Nummer und git den exit Status der Aufgabe aus.

`bbc <--stop|-s> <id>`

10.2.11 Docker Abbild hinzufügen

Fügt ein Docker Abbild als Ausführungsumgebung hinzu. Legt sämtliche Eigenschaften eines Docker Abbildes mit einer DockerFile fest. Beispielsweise Betriebssystem und installierte Programme.

`bbc <--add-image|-ai> <imagename> <Ordner der DockerFile>`

10.2.12 Docker Abbild entfernen

Entfernt ein Docker Abbild als Ausführungsumgebung

`bbc <--remove-image|-ri> <imagename>`

10.2.13 `--block`

Kehrt erst nach der Ausführung der Aufgabe zum Aufrufer zurück. Nützlich um voneinander abhängige Aufgaben, in einem Konsolen Skript, nacheinander auszuführen.

10.2.14 `--priority`

Legt die Priorität der auszuführenden Aufgabe fest, gefolgt von low (4), medium (3), high (2), extreme (1), bananas (0). Es kann der Name bzw. die Zahl in der Klammer als Priorität verwendet werden.

10.2.15 `--min-cpu-count` und `--max-cpu-count`

Legen die minimale bzw. maximale anzahl der verwendbaren CPU fest.

10.2.16 `--min-ram` und `--max-ram`

Legen den minimal verfügbaren bzw. maximale verwendbaren Arbeitsspeicher fest.

11 Testfälle/Testszenarien

11.1 Grundlegende Testfälle

T1 **Verbinden des Clients mit dem Server**

Erklärung Ziel ist zu testen, ob sich der Client beim Programstart automatisch mit dem Server verbinden kann.

Ablauf Ausgegangen wird von einem bereits laufenden System, das mindestens aus dem Server und einem Arbeiter besteht. Der Nutzer versucht eine Aufgabe zu erstellen. Wenn das System funktioniert, bekommt er die Job-Id seiner Aufgabe zurück. Bekommt er die Fehlermeldung

```
Error: Can not find Server
```

so konnte keine Verbindung zum Server hergestellt werden. Erhält er die Fehlermelung

```
Error: Could not authenticate to the Server
```

so konnte der Nutzer sich nicht gegenüber dem Server authentisieren. Erhält er eine andere Fehlermeldung, so konnte die Aufgabe nicht gestartet werden.

T2 **Festlegen von Prioritäten**

Erklärung Ziel ist zu testen, ob Aufgaben mit einer höheren Priorität bevorzugt werden.

Ablauf Ausgegangen wird von einem bereits funktionierenden System mit genau einem Arbeiter, in dem Aufgaben verteilt und bearbeitet werden können. Der Nutzer startet eine Aufgabe die einige Zeit benötigt. Während der Bearbeitung gibt er eine Aufgabe mit einer niedrigen Priorität auf gefolgt von derselben Aufgabe mit einer hohen Priorität. Sind alle drei Aufgaben bearbeitet, startet er wieder eine Aufgabe, die einige Zeit benötigt. Danach erstellt er eine Aufgabe mit einer hohen Priorität und danach dieselbe Aufgabe mit niedriger Priorität. Wenn das System korrekt funktioniert, werden in beiden Fällen die Aufgaben mit hoher Priorität zuerst bearbeitet.

T3 **Festlegen des Betriebssystems**

Erklärung Ziel ist zu testen, ob der Nutzer verschiedene Betriebssysteme angeben kann, die für die Bearbeitung seiner Aufgabe verwendet werden sollen.

Ablauf Ausgegangen wird von einem bereits funktionierenden System mit mehreren Arbeitern und mindestens zwei verschiedenen Arbeitern. Der Nutzer startet eine Aufgabe, die das Betriebssystem ausliest und ausgibt. Diese soll für jedes Betriebssystem einmal eingereiht werden mit einem Parameter, der das jeweilige Betriebssystem anfordert.

T4 **Abfragen eines Aufgabenstatus**

Erklärung Ziel ist zu testen, ob der Nutzer den korrekten Status seiner Aufgaben abfragen kann.

Ablauf Ausgegangen wird von einem funktionierenden System mit genau einem Arbeiter, der Aufgaben entgegennehmen und verarbeiten kann. Der Nutzer reiht mehrere Aufgaben mit unterschiedlichen Prioritäten ein. Die Aufgaben mit niedrigeren Prioritäten sollten in der Warteschlange sein und die Aufgaben mit höherer Priorität sollten entweder gestartet oder weiter vorne in der Warteschlange sein.

T5 **Benachrichtigung bei Abschluss einer Aufgabe**

Erklärung Ziel ist zu testen, ob der Nutzer nach Abschluss einer Aufgabe vom System benachrichtigt wird.

Ablauf Ausgegangen wird von einem funktionierenden System, das in der Lage ist, Aufgaben anzunehmen und zu verarbeiten. Der Nutzer reiht eine Aufgabe ein und prüft regelmäßig den Status seiner Aufgabe. Sollte die Aufgabe abgeschlossen sein, so sollte zeitnah eine Benachrichtigung per E-Mail folgen.

T6 **Erstellen von Sicherungen**

Erklärung Ziel ist zu testen, ob das System regelmäßige Sicherungen seiner Aufgaben anlegt.

Ablauf Ausgegangen wird von einem funktionierenden System, das in der Lage ist, Aufgaben entgegenzunehmen und zu verarbeiten. Es wird eine Aufgabe eingereicht die hinreichend lange bearbeitet wird. Nach Ablauf eines Sicherungsintervalls wird im Dateisystem nach einer Sicherung der Aufgabe gesucht.

T7 **Anforderung der Standard-Ausgabe**

Erklärung Ziel ist zu testen, ob der Nutzer in der Lage ist, sich die korrekte Standardausgabe seiner Aufgabe anzusehen

Ablauf Ausgegangen wird von einem funktionierenden System mit genau einem Arbeiter, das in der Lage ist, Aufgaben entgegenzunehmen und zu verarbeiten. Der Nutzer reiht eine Aufgabe mit bekannter Ausgabe ein. Nach Abschluss der Aufgabe sollte die Ausgabe dieser entsprechen. Zusätzlich sollte der Abruf der Ausgabe einer nicht-existierenden Aufgabe eine Fehlermeldung ausgeben.

11.2 Erweiterte Testfälle

T8 **Abbrechen von zu langen Aufgaben**

Erklärung Ziel ist zu testen, ob der Server in der Lage ist, Aufgaben, die zu viel Zeit benötigen, abzubrechen.

Ablauf Ausgegangen wird von einem funktionierenden System, das in der Lage ist Aufgaben entgegenzunehmen und zu verarbeiten und den Auftraggeber nach Abschluss einer Aufgabe zu benachrichtigen. Der Nutzer reiht eine Aufgabe ein, die nicht beendet. Nach einem festen Zeitintervall sollte der Benutzer eine Benachrichtigung über den erzwungenen Abschluss seiner Aufgabe erhalten.

T9 **Manuelles Stoppen von Aufgaben**

Erklärung Ziel ist zu testen, ob der Nutzer in der Lage ist, seine Aufgaben selbst abzubrechen.

Ablauf Ausgegangen wird von einem funktionierenden System, das in der Lage ist Aufgaben entgegenzunehmen, zu verarbeiten und den Status von Aufgaben zurückzugeben. Der Nutzer reiht eine Aufgabe ein und fordert im Anschluss den Stopp der Aufgabe. Bei einer Statusabfrage sollte die Aufgabe nun als gestoppt angezeigt werden.

T10 **Manuelle Sicherung von Aufgaben**

Erklärung Ziel ist zu testen, ob der Nutzer seine Aufgaben manuell sichern kann.

Ablauf Ausgegangen wird von einem System, das in der Lage ist, Aufgaben entgegenzunehmen und zu bearbeiten. Der Nutzer reiht eine Aufgabe ein und fordert im Anschluss daran die manuelle Sicherung dieser. Im Dateisystem wird nach der Sicherung dieser Aufgabe gesucht.

Glossar

Arbeiter Ein Programm auf einem Rechner, der Aufgaben ausführt.

Aufgabe Jedes Programm (z.B. Skripte, Simulationen, etc.), das ein Benutzer auf einem Rechenknoten ausführen möchte.

Befehlszeile Anwendung, mit der Befehle auf dem Rechner ausgeführt werden.

Benutzer Eine Person, die dazu in der Lage ist, Befehle auszuführen.

Client Ein Programm auf einem Benutzer PC, welches mit dem Server kommuniziert.

CPU Central Processing Unit, Kern jedes Rechners um Anwendungen auszuführen.

Daemon Ein Dienst der Anfragen annimmt und beantwortet.

Datenbank Enthalten verschiedene Informationen, so dass Abfragen nach bestimmten Merkmalen effizient möglich sind.

Docker Ein Programm, mit dessen Hilfe man Aufgaben von dem Host System abkapseln kann.

Docker-Container Ein Container ist eine Standardeinheit von Software, die Code und alle Abhängigkeiten zusammenfasst, damit die Anwendung auf unterschiedlichen Rechnersystemen läuft.

Fehlerbehandlung Vorschriften oder Abläufe, die zur Korrektur eines Fehlers dienen.

Konfigurationsdatei Eine Datei die bestimmte Einstellungen speichert.

Parameter Eine für die Ausführung eines Programms relevante Variable.

Priorität Ein diskretes Maß für Wichtigkeit bzw. Relevanz. Meist eine ganze Zahl, wobei gewisse Werte auch durch vorher definierte Wörter bezeichnet werden können.

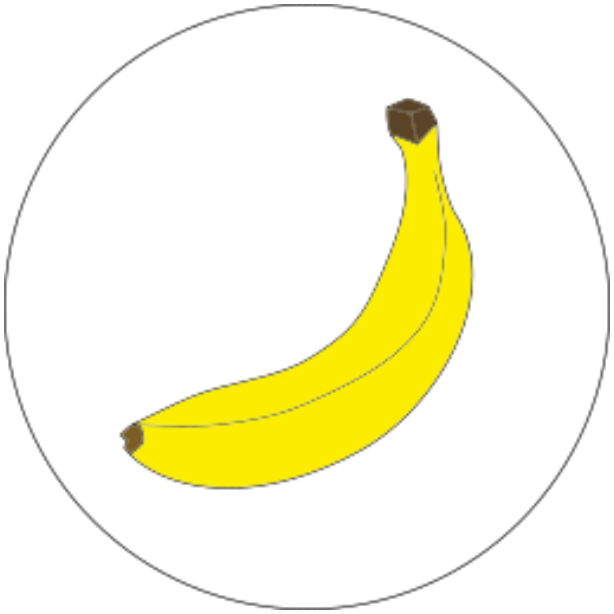
Schnittstelle Eine Art Verbindung zwischen zwei oder mehr voneinander unabhängigen Systemen.

Server Ein Programm auf einem Server, der die Benutzer (Außenwelt) mit den Arbeitern (Privates Netzwerk) verbindet.

Statistik Nützliche Informationen (z.B. Dauer der Task, Statuscode, etc.) über eine Aufgabe, die gesammelt werden sollen.

Warteschlange Datenstruktur, in der Aufgabe für die weitere Verwendung zwischengespeichert werden. Kann mit einer Priorität behaftet sein, welche bestimmt, in welcher Reihenfolge die Aufgabe aus der Warteschlange entfernt werden.

Web-API Web-API (Web-Application Programming Interface) ist eine Schnittstelle über das Internet, die eine Sammlung von Funktionen zur Verfügung stellt, die es Benutzern ermöglichen, auf bestimmte Informationen einer Anwendung, eines Betriebssystems oder anderer Dienste zuzugreifen.



Balanced Banana

A Distributed Task Scheduling System

Entwurf

Niklas Lorenz, Thomas Häuselmann, Rakan Zeid Al Masri, Christopher Lukas Homberger und Jonas Seiler

31. März 2020

Inhaltsverzeichnis

1 Einleitung

Mit diesem Entwurfsdokument möchten wir die Architektur unseres Programms vorstellen und erläutern. Das Dokument zeigt zunächst die Konzeption unseres Systems auf hohem Niveau. Anschließend werden die Besonderheiten der einzelnen Module und die Klassen innerhalb der Module erläutert. Darüber hinaus sind Sequenzdiagramme verfügbar, um die Funktionalität unseres Programms weiter zu veranschaulichen und ein tieferes, praktischeres Verständnis der Funktionsweise zu vermitteln.

2 Aufbau

2.1 Architektur

Mit Rücksicht auf die verschiedenen Aspekte, die zur Verwendung unseres Programms gehören, wurde unser Programm in 3 Blöcke unterteilt. Diese 3 Blöcke arbeiten als jeweils eigenständiges Programm:

- **Die Client Anwendung**, die für die Interaktion mit einem menschlichen Benutzer zuständig ist.
- **Die Server Anwendung**, die den Hauptteil der Logik des Programms ausmacht. Sie ist für die Verteilung der Benutzeraufgaben auf die Arbeiter zuständig, was den Hauptaspekt unseres Programms darstellt.
- **Die Worker Anwendung**, die für die tatsächliche Ausführung der Benutzeraufgaben auf den verfügbaren Arbeiterrechnern zuständig ist.

Zusätzlich existieren weitere Module, die in diesen Anwendungen Verwendung finden.

2.1.1 Kommunikation

So ist das Kommunikation-Modul dafür zuständig, dass sich die Anwendungen untereinander austauschen können und somit der Benutzer in der Lage ist, seine Aufgaben mithilfe der Client Anwendung in Auftrag zu geben und Abfragen zu seinen Aufgaben stellen kann.

Die Netzwerkkommunikation erfolgt stets durch den Austausch von Nachrichten, die einen festen Satz an Daten beeinhalten. Diese Daten werden vom Empfänger verwendet, um eine bestimmte Wirkung zu erzielen. Beispielsweise ist in einer Aufgaben Nachricht die Information hinterlegt, um was für eine Form von Aufgabe es sich handelt, sowie wie diese Aufgabe ausgeführt werden soll. Eine neue Aufgabe im System einzureichen wird somit von Details, wie etwa der Priorität der Aufgabe begleitet.

Die Kommunikation an sich kann prinzipiell auf zwei Wegen erfolgen. Zum einen im Klartext, also ohne Verschlüsselung oder Sicherung der Übertragenen Daten. Diese Variante wird zumeist zwischen Server und Worker verwendet, da diese sich sicher vertrauen können. Bei der Kommunikation zwischen Server und Client hingegen, können verschiedene Sicherheitsprotokolle verwendet werden. Prinzipiell werden Nachrichten über einen SSH Sockel versandt, der von sich aus die Nachrichten gegen Unerwünschte Lauscher schützt. Weiter ist es möglich einen Benutzer mithilfe beispielsweise einer LDAP Authentifizierung zu identifizieren. Somit sind die Benutzer voreinander geschützt und können sich nur schwerer in die Quere kommen.

2.1.2 CommandLineProcessing

Dieses Modul beschäftigt sich mit dem Einlesen von Befehlen auf der Befehlszeile. Das Modul findet Hauptsächlich in der Client Anwendung Verwendung, ist jedoch in allen 3 Anwendungen vorhanden, da diese gelegentlich auch mit der Befehlszeile interagieren müssen (mindestens einmal beim Anwendungsstart).

Innerhalb der Server und Worker Anwendungen wird dieses Modul nur sehr selten nach dem Anwendungsstart verwendet, da Server und Worker neben dem Startbefehl nur wenige oder keine weiteren Befehle von der Befehlszeile entgegennehmen. Viel häufiger ist dies in der Client Anwendung der Fall, bei jeder Interaktion mit dem Benutzer muss das Modul Befehlszeile verwendet werden, um die Intentionen des Benutzers zu verstehen.

Das Modul kann in 2 Schritten zum einen die Befehlsargumente der Befehlszeile einlesen und überprüfen, zum anderen auch vom Benutzer angegebene Standardwerte aus einer lokalen Konfigurationsdatei verwenden. Hierbei werden jedoch immer die Argumente der Befehlszeile den Argumenten der Konfigurationsdatei vorgezogen. Hier kommt ins Spiel, dass alle Anwendungen das Befehlszeilen Modul besitzen: Sie können vorbereitete Aufgabenkonstrukte von anderen Anwendungen entgegennehmen und ergänzen oder auslesen. Daher ist die Server Anwendung in der Lage, globale Standardwerte für fehlende Argumente in dem Befehlsaufruf des Benutzers einzusetzen. Somit muss nicht bei jedem Aufruf der Benutzer daran denken, alle Argumente anzugeben, obwohl er vermutlich immer die gleichen Angeben würde.

2.1.3 Warteschlange

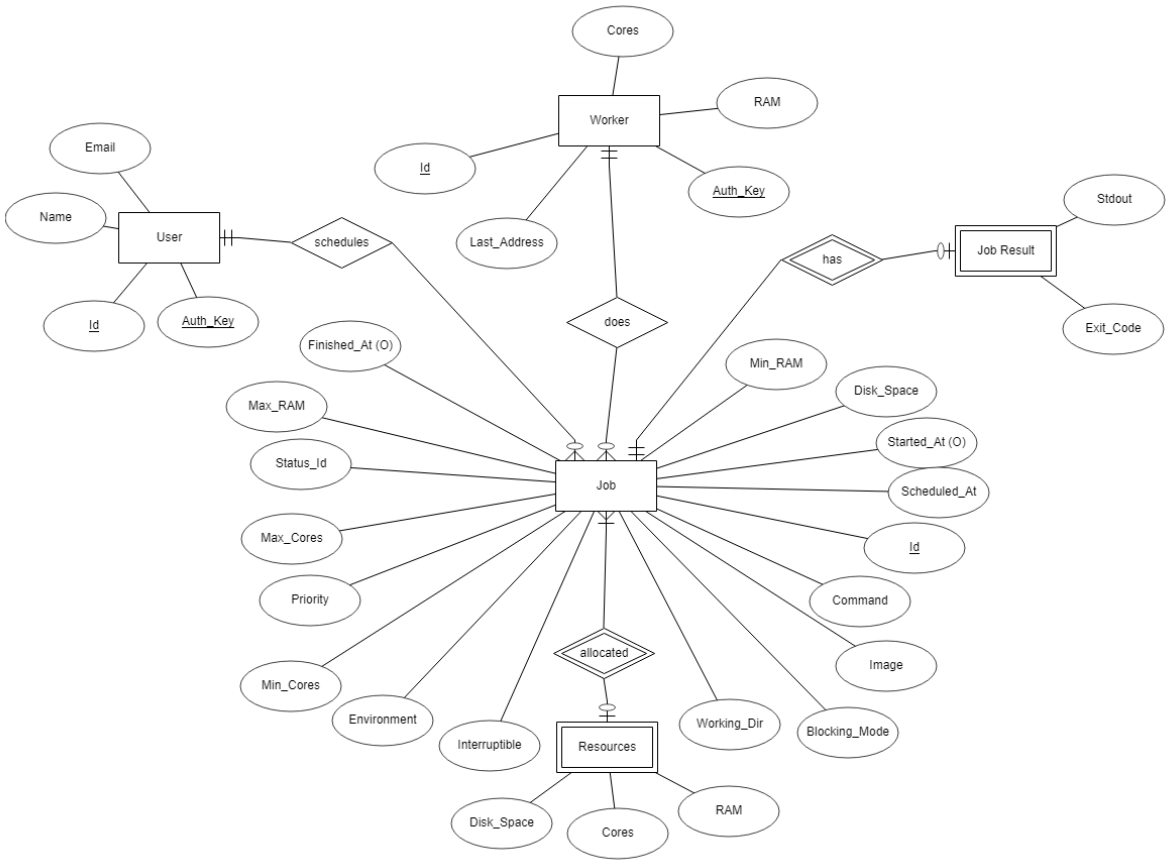
Das Modul Warteschlange realisiert eine Grundlage für die Verteilung von Benutzeraufgaben, die von den Benutzern gesetzte Prioritäten berücksichtigen soll. Die Warteschlange ermöglicht eine effiziente Auswahl der auszuführenden Aufgaben aus dem Pool aller Aufgaben.

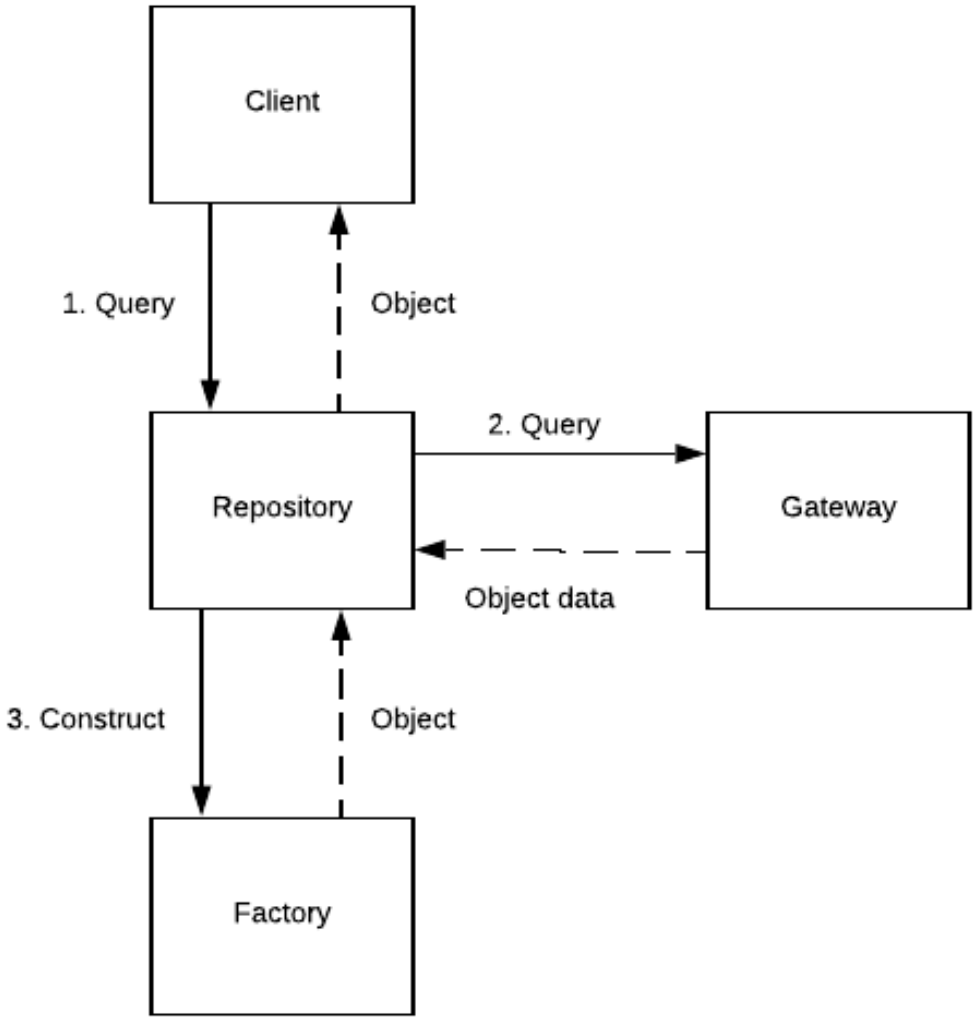
Die Warteschlange ist dabei eine Art Multi Level Queue. Das bedeutet, dass die Warteschlange intern aus mehreren, unabhängigen Warteschlangen besteht. Somit kann die Warteschlange Aufgaben nach Priorität unterscheiden, ohne die Priorität bei jeder Aufgabe explizit zu kennen. Auch ist somit eine Aufwertung einer Priorität einfach möglich.

Die Auswahl der bearbeitenden Aufgaben erfolgt durch die zentrale Einheit der Server Anwendung, dem Scheduler, der aus der Warteschlange genau die Aufgabe zum Berechnen auswählt, die die höchste Priorität besitzt und mit den verfügbaren Arbeiterressourcen ausgeführt werden kann. Aspekte wie Verhungerung von Aufgaben in der Warteschlange werden dabei berücksichtigt. Aufgaben, die nicht zum Zuge kommen, weil stets eine andere Aufgabe mit höherer Priorität vorhanden ist, erfahren nach einer gewissen Wartedauer eine Aufwertung ihrer Priorität. Aufgaben, die stets von niedrig priorisierten, aber weniger anspruchsvollen, Aufgaben überholt werden, können nur eine maximale Anzahl von Überholungen erfahren.

2.1.4 Datenbank

Die Datenbank enthält relevante Daten über verschiedene Entitäten in unserem Programm. Mit Hilfe von SQL-Queries kann unser Programm diese Daten abrufen und sinnvoll nutzen. Unsere Datenbank verwendet ein relationales Datenbank-Managementsystem, um Informationen in verschiedenen Tabellen zu speichern. Darüber hinaus sind die Tabellen über Beziehungen miteinander verbunden, die den Sinn der Daten weiter verdeutlichen. Nachfolgend finden Sie ein Entity-Relationship-Diagramm, das zeigt, wie unsere Datenbank konzipiert ist:

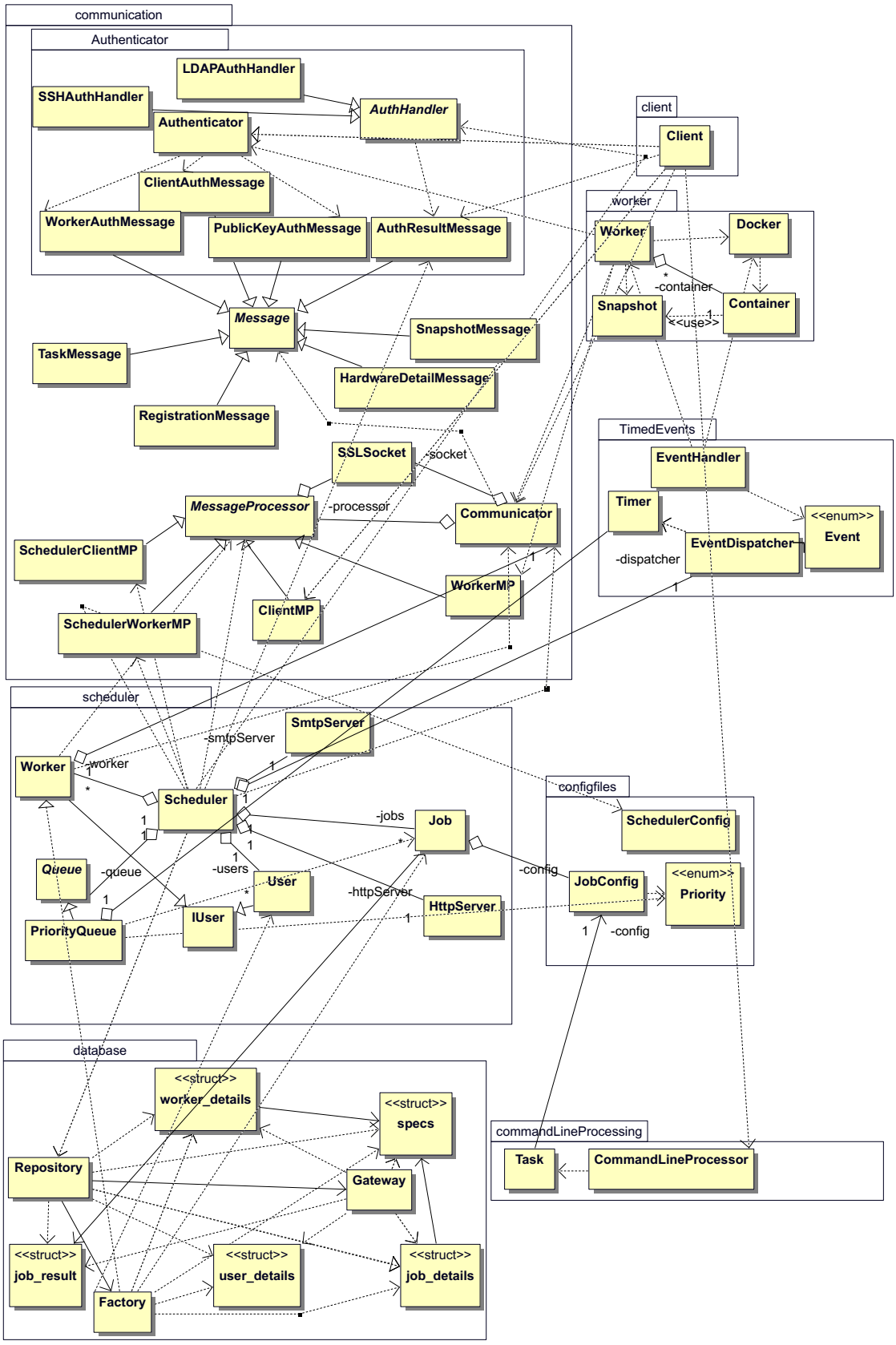




High-Level-Darstellung des Datenbankdesigns

Die Datenbank ist in drei verschiedene Klassen unterteilt: Die Repository-, die Gateway- und die Factory-Klasse. Die Gateway-Klasse verwendet das Qt-Framework, um sich mit der Datenbank zu verbinden und die SQL-Queries durchzuführen. Die Factory-Klasse ist für die Erstellung von Objekten aus den vom Gateway zurückgegebenen Daten verantwortlich. Das Repository nutzt beide Klassen und dient als Schnittstelle für den Zugriff auf die Datenbank.

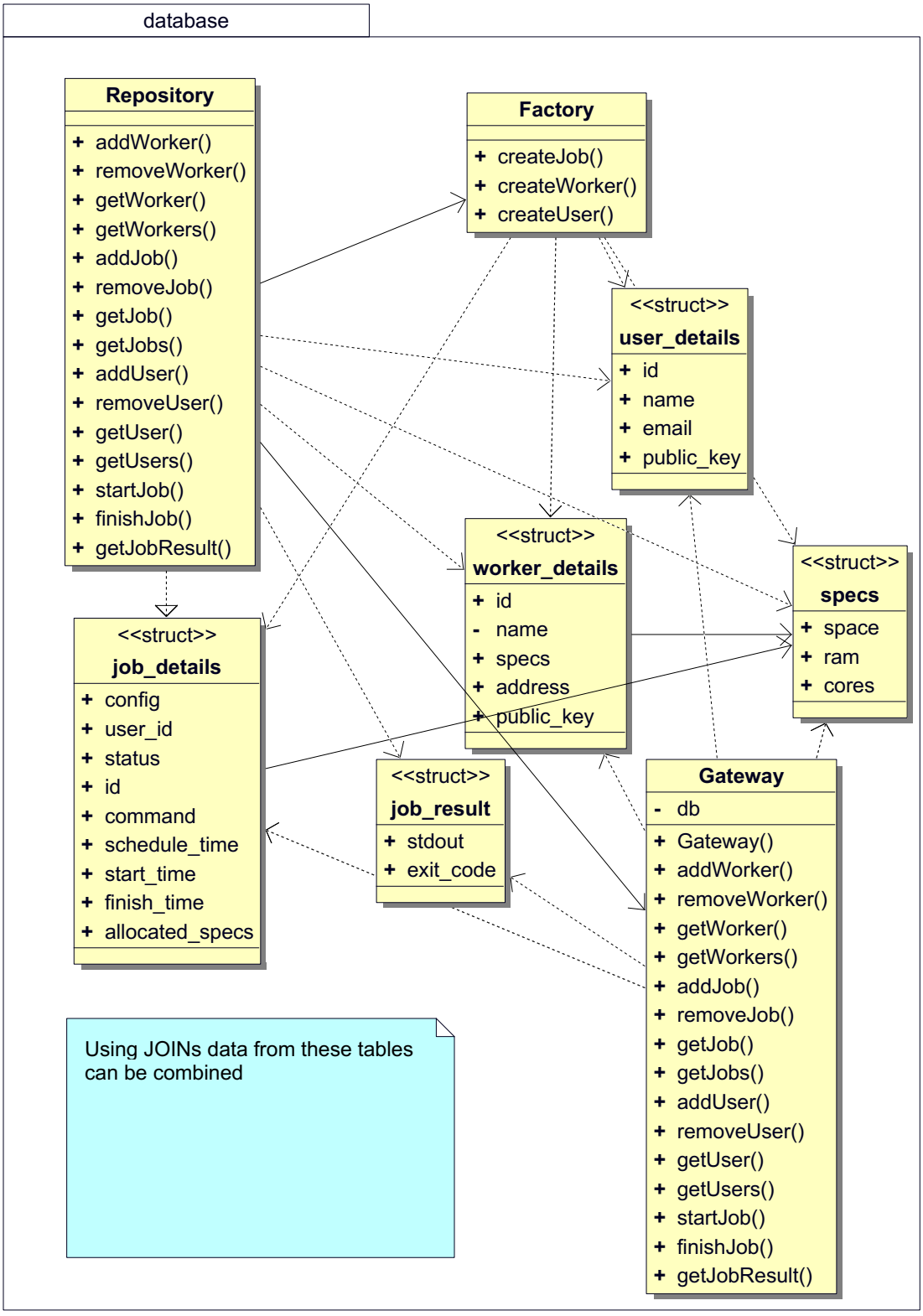
2.2 Klassendiagramm



3 Klassenbeschreibung

3.1 Datenbank

3.1.1 Model



3.1.2 Repository

Die Repository-Klasse ist die Schnittstelle, die der Rest des Programms verwendet, um SQL-Queries durchzuführen und deren Ergebnisse zu interpretieren. Es verwendet die Gateway-Klasse, um die SQL-Queries durchzuführen und erzeugt ein Objekt, indem es der Factory-Klasse die von Gateway zurückgegebenen Daten gibt.

Methoden

- *public uint64_t addWorker(std::string name, std::string public_key, int space, int ram, int cores, std::string address)* Fügt einen Worker zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeWorker(uint64_t id)* Löscht einen Worker aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public Worker getWorker(uint64_t worker_id)* Gibt den Worker mit der angegebenen ID zurück.
- *public std::vector<std::shared_ptr<Worker» getWorkers()* Gibt alle Workers zurück.
- *public uint64_t addJob(uint64_t user_id, JobConfig config, std::chrono::time_point schedule_time, std::string command)* Fügt einen Job zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeJob(uint64_t job_id)* Löscht einen Job aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public Job getJob(uint64_t job_id)* Gibt den Job mit der angegebenen ID zurück.
- *public std::vector<std::shared_ptr<Job» getJobs()* Gibt alle Jobs zurück.
- *public uint64_t addUser(std::string name, std::string email, std::string public_key)* Fügt einen User zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeUser(uint64_t user_id)* Löscht einen User aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public User getUser(uint64_t user_id)* Gibt den User mit der angegebenen ID zurück.
- *public std::vector<std::shared_ptr<User» getUsers()* Gibt alle Users zurück.
- *public bool startJob(uint64_t job_id, uint64_t worker_id, specs specs, std::chrono::time_point start_time)* Aktualisiert den Eintrag eines Jobs in der Datenbank mit einer Startzeit, den zugewiesenen Ressourcen und dem zugeordneten Mitarbeiter. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public bool finishJob(uint64_t job_id, std::chrono::time_point finish_time, std::string stdout, int8_t exit_code)* Aktualisiert den Eintrag eines Jobs mit Endzeit, Ausgabe und Exitcode. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public job_result getJobResult(uint64_t job_id)* Gibt die Ergebnisse eines **fertigen** Jobs zurück.

3.1.3 Gateway

Die Gateway-Klasse verbindet sich mit der Datenbank und führt die SQL-Queries aus.

Attribute

- *QSqlDatabase db* Verwaltet die Verbindung zur Datenbank.

Methoden

- *public uint64_t addWorker(std::string name, std::string public_key, int space, int ram, int cores, std::string address)* Fügt einen Worker zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeWorker(int uint64_t)* Löscht einen Worker aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public worker_details getWorker(uint64_t worker_id)* Gibt die Daten des Workers mit der angegebenen ID zurück.
- *public std::vector<std::shared_ptr<worker_details» getWorkers()* Gibt die Daten jedes Mitarbeiters zurück.
- *public uint64_t addJob(uint64_t user_id, JobConfig config, std::chrono::time_point schedule_time, std::string command)* Fügt einen Job zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeJob(uint64_t job_id)* Löscht einen Job aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public job_details getJob(uint64_t job_id)* Gibt die Daten des Jobs mit der angegebenen ID zurück.
- *public std::vector<std::shared_ptr<job_details» getJobs()* Gibt die Daten jedes Jobs zurück.
- *public uint64_t addUser(std::string name, std::string email, std::string public_key)* Fügt einen User zur Datenbank hinzu und gibt seine ID zurück.
- *public bool removeUser(uint64_t user_id)* Löscht einen User aus dem DB. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public user_details getUser(uint64_t user_id)* Gibt die Daten des Users mit der angegebenen ID zurück
- *public std::vector<std::shared_ptr<user_details» getUsers()* Gibt die Daten jedes Users zurück.
- *public bool startJob(uint64_t job_id, uint64_t worker_id, specs specs, std::chrono::time_point start_time)* Aktualisiert den Eintrag eines Jobs in der Datenbank mit einer Startzeit, den zugewiesenen Ressourcen und dem zugeordneten Mitarbeiter. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public bool finishJob(uint64_t job_id, std::chrono::time finish_time, std::string stdout, int8_t exit_code)* Aktualisiert den Eintrag eines Jobs mit Endzeit, Ausgabe und Exitcode. Gibt true zurück, wenn die Operation erfolgreich war, ansonsten false.
- *public job_result getJobResult(uint64_t job_id)* Gibt die Ergebnisse eines **fertigen** Jobs zurück.

3.1.4 Factory

Erzeugt Objekte aus gegebenen Daten.

Methoden

- *public Job createJob(job_details info)* Erzeugt ein Job-Objekt.
- *public Worker createWorker(worker_details info)* Erzeugt ein Worker-Objekt.
- *public User createUser(user_details info)* Erzeugt ein User-Objekt.

3.1.5 struct `user_details`

Eine struct, die alle relevanten Userdaten kapselt.

Attribute

- *uint64_t id* Eine eindeutige ID zur Identifizierung des Users.
- *std::string name* Der Name des Users.
- *std::string email* Die Email des Users.
- *std::string public_key* Ein öffentlicher Schlüssel, der im Authentifizierungsverfahren Anwendung findet. Wird vom Client bzw. Worker erzeugt.

3.1.6 struct specs

Eine struct, die alle relevanten Hardware-Spezifikationen des Rechners kapselt.

Attribute

- *int space* Speicherplatz.
- *int ram* Random-Access Memory.
- *int cores* Anzahl der CPU-Kerne.

3.1.7 struct `worker_details`

Eine struct, die alle relevanten Workerdaten kapselt.

Attribute

- *uint64_t id* Eine eindeutige ID zur Identifizierung des Workers.
- *specs specs* Hardware-Spezifikationen des Rechners.
- *std::string address*
- *std::string public_key* Ein öffentlicher Schlüssel, der im Authentifizierungsverfahren Anwendung findet. Wird vom Client bzw. Worker erzeugt.

3.1.8 struct `job_details`

Eine struct, die alle relevanten Jobdaten kapselt.

Attribute

- *JobConfig config* Enthält die Konfiguration eines Jobs.
- *uint64_t user_id* Die ID des Users, der diesen Job gescheduled hat.
- *int status* Der Statuscode eines Jobs.
- *uint64_t id* Eine eindeutige ID zur Identifizierung des Jobs.
- *std::string command* Der Befehl des Jobs, der in der Befehlszeile eingegeben wurde.
- *std::chrono::time_point schedule_time* Die Uhrzeit, zu der der Job gescheduled wurde.
- *std::chrono::time_point start_time* Die Uhrzeit, zu der der Job gestartet wurde.
- *std::chrono::time_point finish_time* Die Uhrzeit, zu der der Job beendet wurde.
- *specs allocated_specs* Die Hardware-Ressourcen, die dem Job zugewiesen wurden.

3.1.9 struct `job_result`

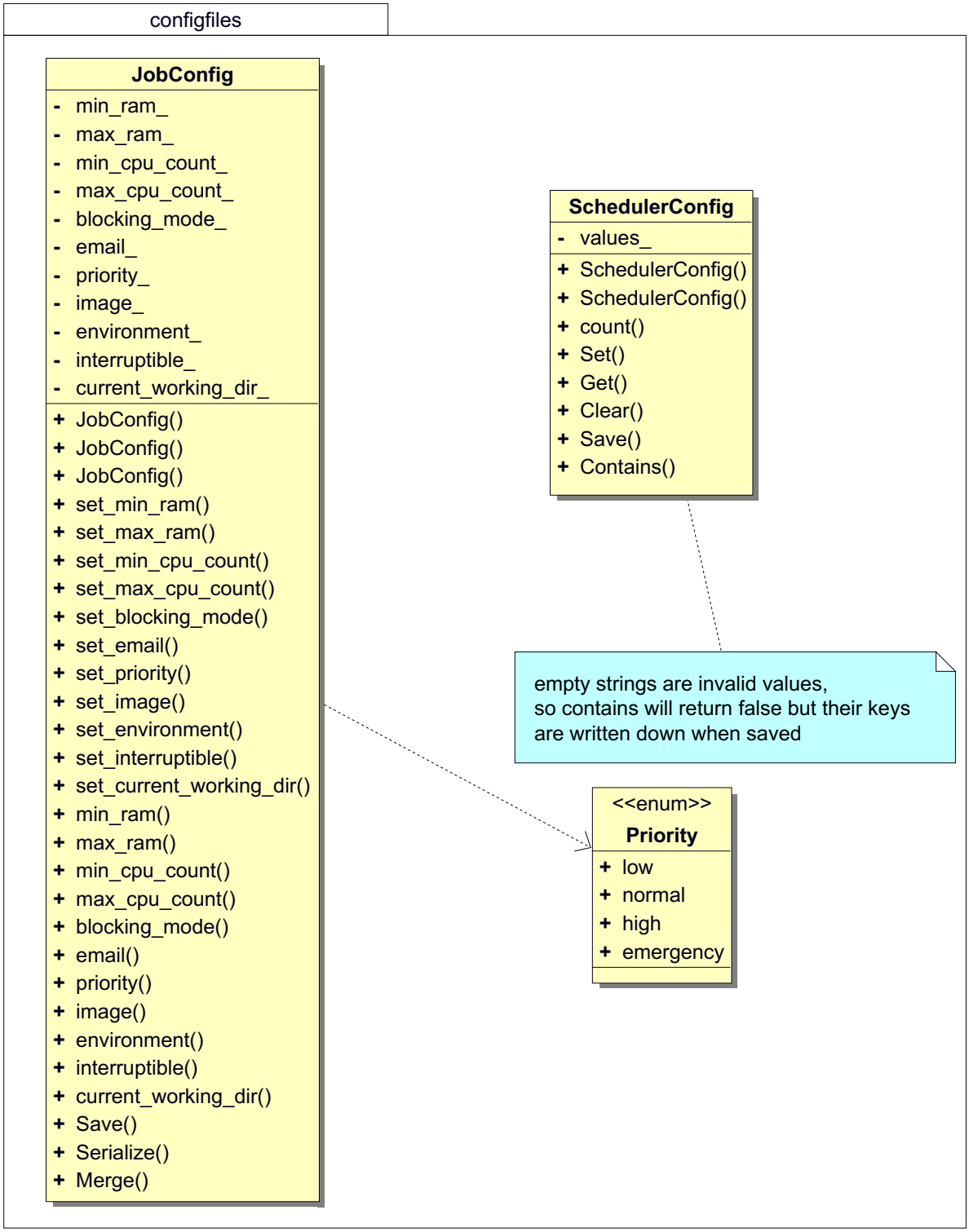
Eine struct, die die Ausgabe und Exit-Code eines Jobs kapselt.

Attribute

- *std::string stdout* Ein Teil der Ausgabe eines Job.
- *int8_t exit_code* Exit-Code eines Jobs.

3.2 Configfiles

3.2.1 Model



3.2.2 enum Priority

Eine Aufzählung aller möglichen Prioritäten, die ein Job haben kann.

Items

- *low* Priorität für Jobs, deren zeitnahe Abarbeitung unwichtig ist.
- *normal* Priorität für Jobs, die zeitnahe abgearbeitet werden sollen.
- *high* Priorität für Jobs, die sehr schnell abgearbeitet werden müssen.
- *emergency* Priorität für Jobs, die umgehend bearbeitet werden müssen. Der Scheduler kann eventuell sogar andere Jobs zeitweise unterbrechen um andere Jobs mit dieser Priorität zu bearbeiten.

3.2.3 JobConfig

Eine Sammlung an allen möglichen Einstellungen, die beschreiben, wie ein Job bearbeitet, bzw. eingeplant werden soll.

Attribute

- *std::optional<uint32_t> min_ram_* Minimal benötigter Hauptspeicher
- *std::optional<uint32_t> max_ram_* Maximal benötigter Hauptspeicher
- *std::optional<uint32_t> min_cpu_count_* Minimal benötigte CPU Kerne
- *std::optional<uint32_t> max_cpu_count_* Maximal zu verwendende CPU Kerne
- *std::optional<bool> blocking_mode_* Ob der Client auf den Abschluss der Aufgabe wartet
- *std::string email_* Zu benachrichtigende Adresse
- *std::optional<Priority> priority_* Priorität des Jobs in der Warteschlange
- *std::string image_* Zu verwendendes Docker Image
- *std::optional<std::vector<std::string» environment_* Liste an Umgebungsvariablen
- *std::optional<bool> interruptible_* Ob der Job unterbrechbar ist.
- *std::optional<std::filesystem::path> current_working_dir_* Verzeichnis, von dem aus der Job im Container ausgeführt werden soll.

Methoden

- *JobConfig()* Erstellt eine leere JobConfig
- *JobConfig(std::stringstream &)* Deserialisiert eine JobConfig aus dem übergebenen Stream
- *JobConfig(std::filesystem::path &)* Lädt eine serialisierte JobConfig aus dem Dateisystem an dem angegebenen Pfad.
- *void set_min_ram(std::optional<uint32_t>)* Setter für min_ram_
- *void set_max_ram(std::optional<uint32_t>)* Setter für max_ram_

- *void set_min_cpu_count(std::optional<uint32_t>)* Setter für min_cpu_count_
- *void set_max_cpu_count(std::optional<uint32_t>)* Setter für max_cpu_count_
- *void set_blocking_mode(std::optional<bool>)* Setter für blocking_mode_
- *void set_email(std::string &)* Setter für email_
- *void set_priority(std::optional<Priority>)* Setter für priority_
- *void set_image(std::string &)* Setter für image_
- *void set_environment(std::optional<std::vector<std::string» &)* Setter für environment_
- *void set_interruptible(std::optional<bool>)* Setter für interruptible_
- *void set_current_working_dir(std::optional<std::filesystem::path> &)* Setter für current_working_dir_
- *std_optional<uint32_t> min_ram()* Getter für min_ram_
- *std_optional<uint32_t> max_ram()* Getter für max_ram_
- *std_optional<uint32_t> min_cpu_count()* Getter für min_cpu_count_
- *std_optional<uint32_t> max_cpu_count()* Getter für max_cpu_count_
- *std_optional<bool> blocking_mode()* Getter für blocking_mode_
- *std::string &email()* Getter für email_
- *std_optional<Priority> priority()* Getter für priority_
- *std::string &image()* Getter für image_
- *std_optional<std::vector<std::string» &environment()* Getter für environment_
- *std_optional<bool> interruptible()* Getter für interruptible_
- *std_optional<std::filesystem::path> &current_working_dir()* Getter für current_working_dir_
- *void Serialize(std::stringstream &)* Schiebt die serialisierte JobConfig in den angegebenen Stream.
- *void Save(std::filesystem::path &)* Speichert die JobConfig im Filesystem an der angegebenen Stelle.
- *void Merge(JobConfig &)* Füllt diese JobConfig mit Werten aus der übergebenen JobConfig auf. In beiden Configs vorhandene Werte werden nicht überschrieben.

3.2.4 SchedulerConfig

Wrapper für eine Hashmap, der auch eine Möglichkeit zum Speichern und Laden ins Filesystem bietet und als Speicher für verschiedene Regeln beim Einplanen der Jobs dient.

Attribute

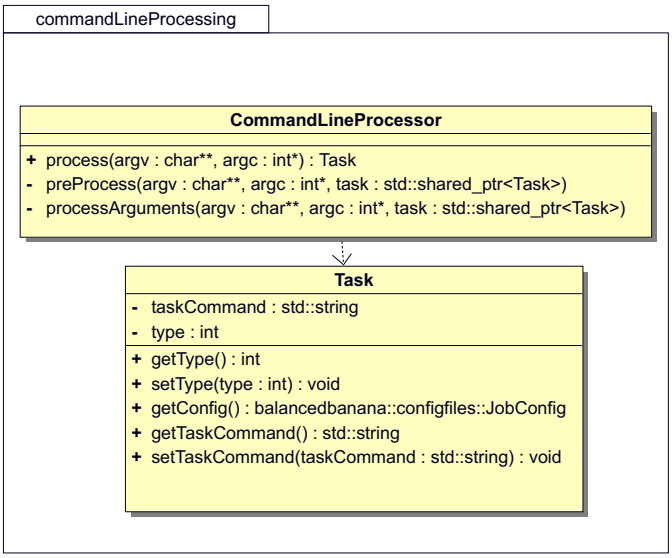
- *std::map<std::string, std::string> values_* Hashmap, die alle Regeln enthält.

Methoden

- *SchedulerConfig()* Erzeugt eine leere SchedulerConfig
- *SchedulerConfig(std::filesystem::path &)* Lädt eine serialisierte SchedulerConfig aus dem Filesystem
- *size_t count()* Gibt an wie viele Elemente sich in der Config befinden.
- *Get(std::string &)* Gibt das Element zurück, das der übergebenen Regel zugeordnet wurde.
- *Set(std::string &key, std::string &value)* Setzt den Wert der Regel mit dem übergebenen key auf den übergebenen Wert.
- *Save(std::filesystem::path &)* Schreibt die Config in das Filesystem an die angegebene Stelle.
- *Contains(std::string &)* Gibt an, ob sich in der Config eine Regel mit dem angegebenen key befindet.

3.3 CommandLineProcessing

3.3.1 Model



3.3.2 CommandLineProcessor

Der Command Line Processor (CLP) ist dafür zuständig, die Benutzereingabe auf der Befehlszeile in eine für das Programm verwendbare Struktur zu überführen. Es werden die Argumente auf der Befehlszeile eingelesen und mit vom Benutzer in einer Konfigurationsdatei angegebenen Standardwerten ergänzt. Anhand der Angaben auf der Befehlszeile kann erkannt werden, um welchen Typ von Befehl es sich handelt, und welche Parameter für die Ausführung des Befehls vorhanden sein müssen.

Methoden

- *public Task process(char** argv, int* argc)* Verarbeitet Argumente der Befehlszeile. Weist jedem Argument zuerst den auf der Befehlszeile spezifizierten Wert, dann den vom Benutzer in einer Konfigurationsdatei hinterlegten Wert zu. Argumente, die nicht auf der Befehlszeile oder in der Konfigurationsdatei vorhanden sind, werden von dem Server mit Standardwerten verfollständigt.

  argv: Array der Bezeichner sowie Werte der Argumente
  argc: Anzahl der Einträge in argv

- *private void preProcess(char** argv, int* argc, std::shared_ptr<Task> task)* Baut aus den übergebenen Argumenten einen Task auf.
  Bestimmt den Typ der Anfrage (neue Aufgabe oder vorhandene Bearbeiten)
  Hierbei werden komplexe Argumente von den eigentlichen Argumenten getrennt (z.B. wird der Aufgaben Startbefehl von den Argumenten unseres Programms abgetrennt, damit dies nicht zu Problemen in späteren Verarbeitungsschritten führt).
  argv und argc werden dementsprechend angepasst.

  argv: Array der Bezeichner und Werte der Argumente
  argc: Anzahl der Einträge in argv
  task: Bekommt einen Aufgaben-Typ zugewiesen

- *private void processArguments(char** argv, int* argc, std::shared_ptr<Task> task)* Wertet die Argumente der Befehlszeile aus und speichert sie in task. Zusätzlich werden die Argumente aus der lokalen Konfigurationsdatei übernommen.

  argv: Array der Bezeichner und Werte der Argumente
  argc: Anzahl der Einträge in argv
  task: Enthält nach Abarbeitungsende alle auf der Befehlszeile eingelesenen Argumente sowie alle Standardargumente der lokalen Konfigurationsdatei

3.3.3 Task

Ein Task ist eine Repräsentation eines auf der Befehlszeile eingegebenen Befehls. Er enthält Informationen, die für die Ausführung des gewünschten Befehls wichtig sind. Dazu gehören im wesentlichen Befehlstyp sowie Argumente.

Attribute

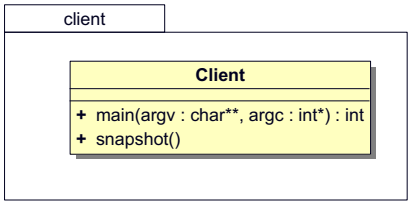
- *taskCommand* Startbefehl für die eingereichte Aufgabe
- *config* Konfiguration dieses Befehls. Argumente der Befehlszeile und der lokalen Konfigurationsdatei.
- *type* Gibt an, um was für eine Form von Anfrage es sich handelt (Starte neue Aufgabe oder frage etwas über eine existierende Aufgabe ab)

Methoden

- *public int getType()* Gibt den Befehlstyp type zurück.
- *public void setType(int type)* Setzt den Befehlstyp zu dem Angegebenen Typ.
- *public JobConfig getConfig()* Gibt Zugang zur Konfiguration des Befehls. Die Konfiguration kann von außerhalb beliebig modifiziert werden.
- *public std::string getTaskCommand()* Gibt den vom Benutzer spezifizierten Startbefehl der eingereichten Aufgabe (falls eine eingereicht wurde) so, wie er auf der Befehlszeile ausgeführt werden soll.
- *public void setTaskCommand(const std::string& taskCommand* Setzt den Startbefehl der eingereichten Aufgabe (falls eine eingereicht wurde).

3.4 Client

3.4.1 Model



3.4.2 Client

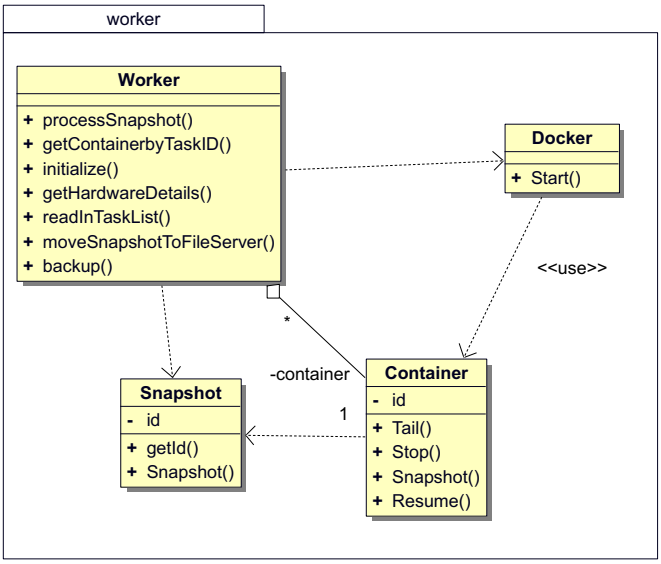
Die Hauptklasse der Client Anwendung. Verarbeitet die Befehls des Benutzers und leitet diese weiter an den Server. Bei Bedarf kann auf eine Rückmeldung des Servers gewartet werden, die dem Benutzer auf der Befehlszeile angezeigt wird.

Methoden

- *int main(char** argv, int* argc)* Wandelt den Befehl des Benutzers mithilfe des CommandLineProcessor in verwendbare Struktur um. Leitet diesen dann an den Server weiter. Bei Bedarf wird die Netzwerkkommunikation offen gehalten, bis der Server eine Rückmeldung gesendet hat. Nach Ausgabe der Rückmeldung ist der Befehl beendet und die Client Anwendung wird beendet.

3.5 Worker

3.5.1 Model



3.5.2 Docker

Methoden

- *public static Container Start(int userid, Task task)* Startet einen Task auf dem Worker mithilfe von Docker. Gibt ein Container Object zurück welches diesen Task während der Ausführung beschreibt.

3.5.3 Container

Attribute

- *std::string id* Docker ID des laufenden Containers.

Methoden

- *public std::string Tail(int lines)* Gibts die letzten lines Zeilen als Zeichenfolge zurück.
- *public void Stop()* Stoppt den Container und gibt Ressourcen frei, kann mit Resume fortgesetzt werden.
- *public Snapshot Snapshot(bool stop)* Erstellt ein Snapshot (Sicherung) des Containers, stoppt ihn danach falls gewünscht.
- *public void Resume(Snapshot snap)* Setzt das Snapshot snap (Sicherung) des Containers wieder her und setzt die instanz fort.

3.5.4 Snapshot

Klasse um ein Docker Snapshot zu verwalten.

Attribute

- *std::string id* Docker ID des laufenden Snapshots.

3.5.5 Worker

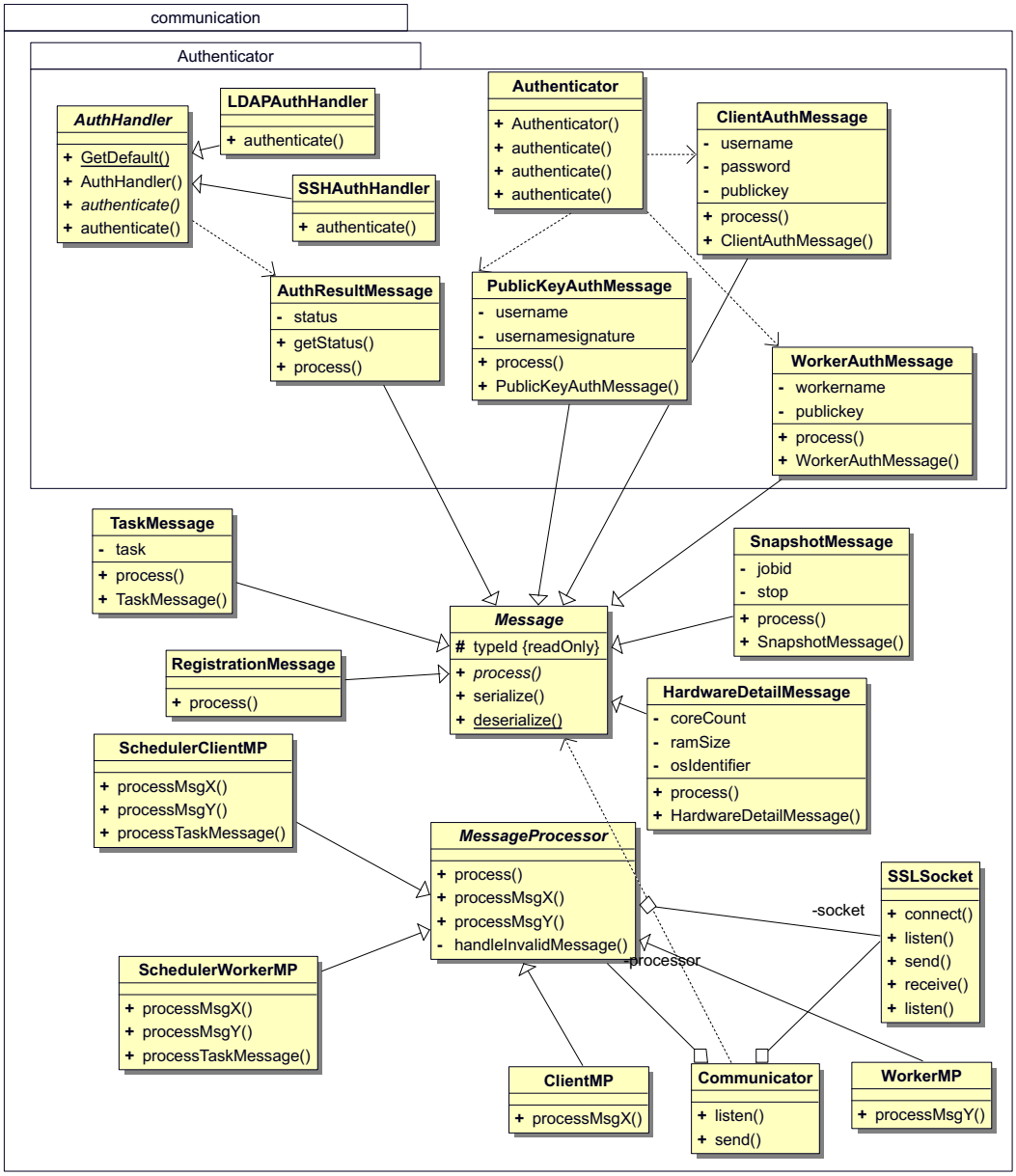
Hauptklasse der Worker instanz.

Methoden

- *public void processSnapshot(Message msg)* Gibts die letzten lines Zeilen als Zeichenfolge zurück.
- *public Task getContainerbyTaskID(uint64_t tid)* Gibt dem zum Task gehörenden Container zurück.
- *public void initialize()* Initialisiert den Worker zur Ausführung der Tasks.
- *public void getHardwareDetails()* Ermittelt CPU Kern Anzahl, RAM, Betriebssystem.
- *public void readInTaskList(std::string pathToTaskList)* Liest Einträge aus einer Datei und interpretiert diese als Aufgaben
- *public void moveSnapshotToFileServer(Snapshot snap, std::string location)* Speichert den Snapshot auf dem File Server, damit ein Benutzer darauf zugriff bekommt. location: wo auf dem FileServer soll der Snapshot gespeichert werden.
- *public void backup(std::string location)* Speichert einen Snapshot aller laufenden Aufgaben an der angegebenen Stelle im FileServer oder nur auf dem Worker, falls location keine gültige Stelle auf dem File Server ist.

3.6 Communication

3.6.1 Model



3.6.2 AuthHandler

Beschreibt Methoden zur Ausführung von Authentifizierungsversuchen auf dem Scheduler. Kann mithilfe verschiedener Protokolle eine Person authentifizieren. Dient der Sicherheit vor Verwechselung und Täuschung.

Methoden

- *public AuthHandler GetDefault()* Gibt einen Allzweck AuthHandler zurück, der in jeder Situation funktioniert.
- *public AuthHandler()* Kontruktor des abstrakten AuthHandler kann nur von Unterklassen aufgerufen werden.
- *public virtual void authenticate(std::shared_ptr<IUser>, std::string password)* Authentifiziere einen Benutzer mithilfe eines Passworts.
- *public void authenticate(std::shared_ptr<IUser>, std::string signature)* Authentifiziere einen Benutzer mithilfe einer Signatur (nicht notwendigerweise ein Passwort)

3.6.3 SSHAuthHandler

Implementiert einen AuthHandler, der mit dem localen OpenSSH client arbeitet.

Methoden

- *public void authenticate(std::string username, std::string password)* Implementiert die SSH Authentifizierung eines Benutzers mithilfe eines Passworts.

3.6.4 LDAPAuthHandler

Implementiert einen AuthHandler, der mit der OpenLDAP C Api arbeitet.

Methoden

- *public void authenticate(std::string username, std::string password)* Implementiert die LDAP Authentifizierung eines Benutzers mithilfe eines Passworts.

3.6.5 ClientAuthMessage

Nachricht für die Authentifizierung eines Clienten (Anwendung). Wird beim ersten Verbindungsaufbau verwendet. Bei weiteren Verbindungen können andere Nachrichten Typen erforderlich sein.

Attribute

- *std::string username* Der Name des zu authentifizierenden Benutzers.
- *std::string password* Klartext des vom Benutzer angegebenen Passworts.
- *std::string publickey* Ein öffentlicher Schlüssel, der im Authentifizierungsverfahren Anwendung findet. Wird vom Client bzw. Worker erzeugt.

Methoden

- *public ClientAuthMessage(std::string username, std::string password, std::string pubkey)*
- *public void process(std::shared_ptr<MessageProcessor> mp)*

3.6.6 PublicKeyAuthMessage

Nachricht für die Authentifizierung eines Benutzers mittels eines public Key Verfahrens.

Attribute

- *std::string username* Der Name des zu authentifizierenden Benutzers.
- *std::string usernamesignature* Eine Signatur, die mithilfe des private keys vom Client bzw. Worker erzeugt wird. Kann vom Scheduler mithilfe des zuvor übermittelten public keys überprüft werden.

Methoden

- *public void process(std::shared_ptr<MessageProcessor> mp)*
- *public PublicKeyAuthMessage(std::string username, std::string usernamesignature)*

3.6.7 AuthResult

Ergebnis der Authentifizierung. Gibt im wesentlichen den Erfolg der Authentifizierung an.

Attribute

- *unsigned long status* Gibt an, wie die Authentifizierung ausgegangen ist. (Erfolg(=0)/Misserfolg(!=0))

Methoden

- *public unsigned long getStatus()*

3.6.8 WorkerAuthMessage

Nachricht, die zur Authentifizierung eines Arbeiters dient. Hier wird im Allgemeinen kein komplexes Sicherheitsverfahren verwendet.

Attribute

- *std::string workername* Name des Arbeiters, der authentifiziert wird.
- *std::string publickey* Ein public Key der in gleichnamigen Verfahren Anwendung findet.

Methoden

- *public void process(std::shared_ptr<MessageProcessor> mp)*
- *public WorkerAuthMessage(std::string workername, std::string pubkey)*

3.6.9 Authenticator

Authentifiziert einen Client und Worker gegenüber dem Scheduler.

Methoden

- *public Authenticator(Communicator comm)* Kontruktor um einen Authenticator für den Communicator comm zu erstellen.
- *void authenticate(std::string username, std::string password, std::string pubkey)* Authentifiziert einen Client bei der ersten Verbindung mit einen Benutzernamen, Passwort und einen RSA Public Key. Der private Key wird auf dem Client aufbewart und nicht versendet, um sich beim nächsten mal ohne Passwort anmelden zu können.
- *void authenticate(std::string workername, std::string pubkey)* Authentifiziert einen Worker bei der ersten Verbindung mit einen vom Worker generierten eindeutigen (Worker-)Namen und einen RSA Public Key
- *void authenticate(std::string username, std::string usernamesignature)* Authentifiziert einen Client / Worker mit Benutzernamen und einer signature, die mit den private Key vom Client automatisch erzeugt wurde (Passwortlose Anmeldenung)

3.6.10 Communicator

Haupteinheit zur Kommunikation der verschiedenen Anwendungen. Ermöglicht senden und empfangen von Nachrichten, die von und zu anderen Einheiten des Systems gesendet werden. Verbirgt die Details der Netzwerkkommunikation vor anderen Teilen der Programme. Wird identisch in allen Anwendungen verwendet.

Methoden

- *public void listen(std::function<void(std::shared_ptr<Message>)> callback)* Warte auf eine eingehende Nachricht. Ruft dann callback auf, mit der eingegangenen Nachricht als Parameter.
- *public void send(Message message)* Sendet eine Nachricht an einen eindeutigen Empfänger.

3.6.11 SSLSocket

Basis der verschlüsselten Kommunikation. Stellt Funktionalität zur sicheren Netzwerkkommunikation mithilfe des SSH Protokolls bereit.

Methoden

- *public void connect(std::string address* Verbindet sich mit einem Socket an der angegebenen Netzwerkadresse.
- *public void listen(unsigned short port, std::function<void(std::shared_ptr<SSLSocket>)> callback)* Wartet auf eine eingehende Nachricht an einem bestimmten Port. Ruft dann eine callback Methode auf mit dem relevanten Socket als Parameter. Die Callback Routine kann dann die Nachricht verarbeiten.

- *public void send(const char* msg)* Sendet eine Nachricht an einen zuvor spezifizierten Empfänger.
- *public void receive(char* data), size_t size)* Nehme eine Netzwerknachricht entgegen, die in Form eines byte Streams am Netzwerkadapter ankam.
- *public void listen(unsigned short port, std::function<void()> callback)* Wartet auf eine eingehende Nachricht und ruft dann eine callback methode auf, die dieses Ereigniss verarbeitet.

3.6.12 MessageProcessor

Haupteinheit für die Verarbeitung eingegangener Nachrichten. Ein Message Processor kann den Typ einer Nachricht identifizieren und je nach Nachrichtentyp die benötigten Prozesse anstoßen, um die Nachricht zu verarbeiten.

Methoden

- *public void process(std::shared_ptr<Message> msg)* Verarbeitet eine engegangene Nachricht, indem deren Typ identifiziert wird und mit den enthaltenen Daten Ereignisse angestoßen werden.
- *private void handleInvalidMessage(std::shared_ptr<Message> msg)* Sollte eine ungültige Nachricht eingegangen sein, wird diese an dieser Stelle behandelt. Kann von der konreten Implementierung des Message Processors abhängig sein.

3.6.13 ClientMP

Message Processor für den Client. Hiermit verarbeitet die Client (Benutzer) Anwendung Nachrichten vom Server, wie zum Beispiel Rückmeldungen auf Statusabfragen.

Methoden

- *public void process(std::shared_ptr<Message> msg)*

3.6.14 SchedulerClientMP

Message Processor für den Server (Scheduler) der Nachrichten verarbeitet, die von Seiten einer Client (Benutzer) Anwendung kommen.

Methoden

- *public void process(std::shared_ptr<Message> msg)*

3.6.15 SchedulerWorkerMP

Message Processor für den Server (Scheduler) der Nachrichten verarbeitet, die von Seiten einer Worker Anwendung kommen.

Methoden

- *public void process(std::shared_ptr<Message> msg)*

3.6.16 WorkerMP

Message Processor für den Worker der Nachrichten verarbeitet, die von Seiten einer Server (Scheduler) Anwendung kommen.

Methoden

- *public void process(std::shared_ptr<Message> msg)*

3.6.17 Message

Basis jeder Nachricht, die von den Anwendungen aneinander versendet werden können. Erzwingt die Möglichkeit zur Eindeutigen Identifikation einer Nachricht anhand einer Id sowie die Möglichkeit eine Nachricht Netzwerkfähig zu machen und aus einer Netzwerknachricht zu rekonstruieren.

Attribute

- *protected const unsigned int typeId* Eindeutige Nummer, die eine Nachricht identifiziert. Wird verwendet um verschiedene Nachrichtentypen mit unterschiedlichen Daten zu unterscheiden.

Methoden

- *public void process(std::shared_ptr<MessageProcessor> mp)* Verarbeite diese Nachricht mithilfe des angegebenen Message Processors.
- *public std::string serialize()* Wandle die Nachricht in eine Abfolge von bytes um, die über das Netzwerk versendet werden kann.
- *public std::shared_ptr<Message> deserialize(const char* msg, unsigned int size)* Rekonstruieren die Nachricht aus einer folge von bytes, die über das Netzwerk angekommen sind.

3.6.18 SnapshotMessage

Nachricht, die das Erstellen eines Snapshots einer laufenden Aufgabe auf einem Worker anfordert. Wird vom Scheduler an einen Worker versendet.

Attribute

- *unsigned long jobid* Welche Aufgabe soll gesichert werden?
- *bool stop* Soll während dem Erstellen des Snapshots die Ausführung der Aufgabe pausiert werden?

Methoden

- *public void process(std::shared_ptr<MessageProcessor>)*
- *public SnapshotMessage(unsigned long jobid, bool stop)*

3.6.19 RegistrationMessage

Nachricht, um sich bei einer anderen Anwendung anzumelden. Wird primär zwischen Server und Worker verwendet, damit die Worker wissen wie sie einen Scheduler ansprechen können und der Scheduler alle verfügbaren Worker kennen kann.

Methoden

- *public void process(std::shared_ptr<MessageProcessor>)*

3.6.20 HardwareDetailMessage

Nachricht, die Informationen über die Hardware eines Workers enthält. Wird vom Worker zum Scheduler gesendet.

Attribute

- *int coreCount* Wie viele CPU Kerne können für eine Aufgabe verwendet werden.
- *int ramSize* Wie viel Arbeitsspeicher ist auf dem Worker vorhanden.
- *std::string osIdentifier* Welches Betriebssystem verwendet der Worker.

Methoden

- *public void process(std::shared_ptr<MessageProcessor>)*
- *public HardwareDetailMessage(int coreCount, int ramSize, std::string osIdentifier)*

3.6.21 TaskMessage

Nachricht, die Informationen zu einer Aufgabe übermittelt. Wird von Client zu Server gesendet.

Attribute

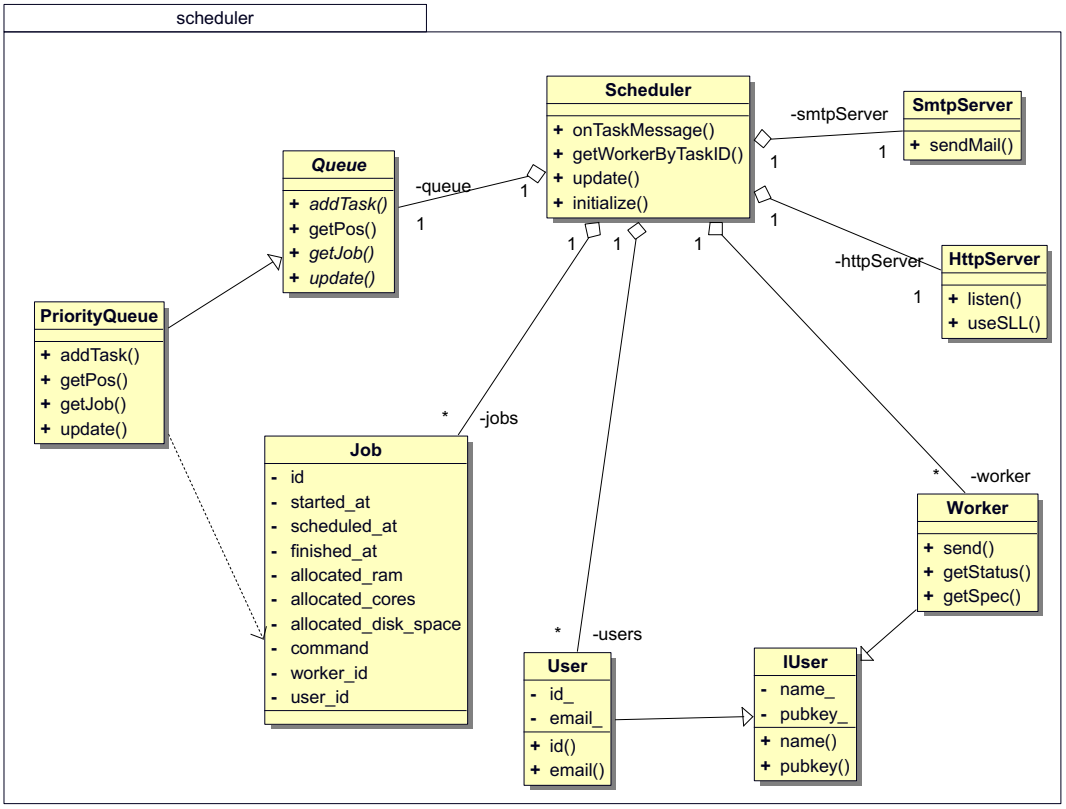
- *Task task* Objekt, welches Informationen über den vom Benutzer angeforderten Befehl enthält. Siehe Task.

Methoden

- *public void process(std::shared_ptr<MessageProcessor>)*
- *public TaskMessage(Task task)*

3.7 Scheduler

3.7.1 Model



3.7.2 Queue

Die Oberklasse einer Warteschlangen-Implementierung. Hier werden alle nötigen Methoden für eine spezielle Implementierung einer Warteschlange definiert, sowie Default-Rückgabewerte falls diese nicht implementiert wurden. Eine spezielle Warteschlange soll als konkrete Strategie diese Klasse erweitern.

Methoden

- *void addTask(Job job)* Fügt der Warteschlange ein Job hinzu. Standardmäßig werden hier Jobs nur mit angefragter ID entfernt.
- *int getPos(int id)* Gibt die Position des Tasks mit der übergebenen ID aus. Standardmäßig wird hier 1äusgegeben.
- *Job getJob(int id)* Gibt den Job mit der angegeben ID aus der Warteschlange heraus. Dies ist eine Fallback-Methode und sollte in allen Warteschlangen-Strategien ersetzt werden.
- *void update()* Wird regelmäßig vom Scheduler aufgerufen. Standardmäßig passiert nichts. Diese Methode soll von einer konkreten Strategie für regelmäßige Checks benutzt werden.

3.7.3 PriorityQueue

Eine spezielle Implementierung einer Warteschlange. Sortiert Tasks nach Prioritäten.

Methoden

- *void addTask(Job job)* Fügt der Warteschlange ein Job hinzu. Dieser wird entsprechend seiner Priorität weiter vorne eingestuft je wichtiger.
- *int getPos(int id)* Gibt die Position des Tasks mit der übergebenen ID aus.
- *Job getJob(int Ram, int Cores)* Gibt den wichtigsten Job, also den mit der höchsten Priorität, aus, der zu den übergebenen Spezifikationen passt.
- *void update()* Überprüft ob ein Job bereits 24 Stunden in der Warteschlange verbracht hat und erhöht gegebenenfalls seine Priorität.

3.7.4 Scheduler

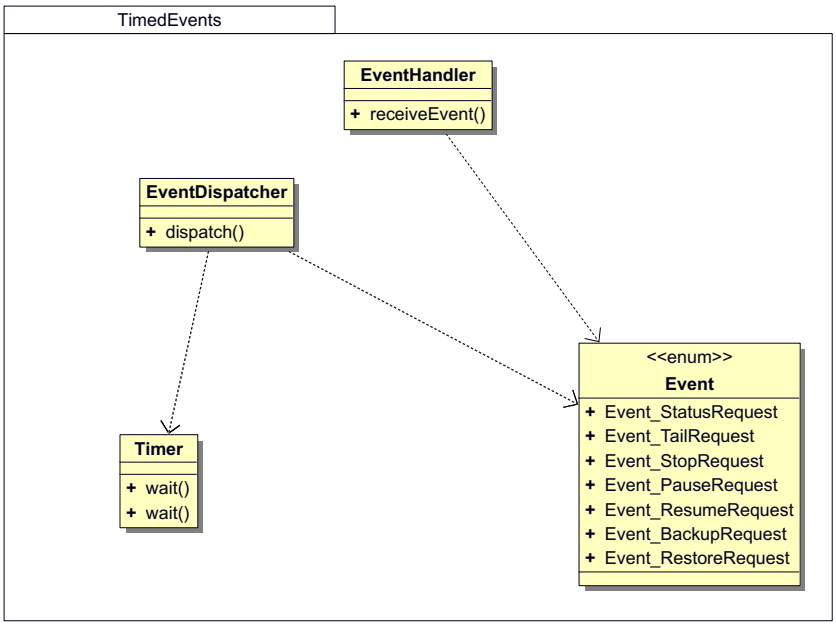
Die Hauptklasse des Schedulers. Hierdurch werden Befehle und Datenbankzugriffe gesteuert.

Methoden

- *void onTaskMessage(Message message)* Verarbeitet den erhaltenen Befehl entsprechend und führt den Inhalt aus.
- *void update()* Prüft alle Worker nach Verbindung und Status und updated gegebenfalls andere Module
- *void intialize()* Stellt Verbindungen zu Workern her und versucht einen Status aus der Datenbank herzustellen.

3.8 TimedEvents

3.8.1 Model



3.8.2 EventHandler

Methoden

- *public void receiveEvent(Event, uint64_t taskID)* Behandelt das angekommene Event.

3.8.3 Timer

Methoden

- *public void wait(int s)* Blockiert den Aufrufer für eine Anzahl an Sekunden
  s: Anzahl der zu wartenden Sekunden
- *public void wait(const std::function<void()>& func)* Wartet s Sekunden und führt dann die Funktion func aus. Der Aufrufer wird nicht blockiert
  s: Anzahl der zu wartenden Sekunden, func: Die auszuführende Funktion

3.8.4 Event

Member

- *Event_StatusRequest* Event einer Statusabfrage
- *Event_TailRequest* Event einer letze Ausgabezeilenabfrage
- *Event_StopRequest* Event zum stoppen einer Aufgabe
- *Event_PauseRequest* Event zum pausieren einer Aufgabe
- *Event_ResumeRequest* Event zum fortsetzen einer Aufgabe
- *Event_BackupRequest* Event zum sichern einer Aufgabe
- *Event_RestoreRequest* Event zum wiederherstellen einer Aufgabe

3.8.5 EventDispatcher

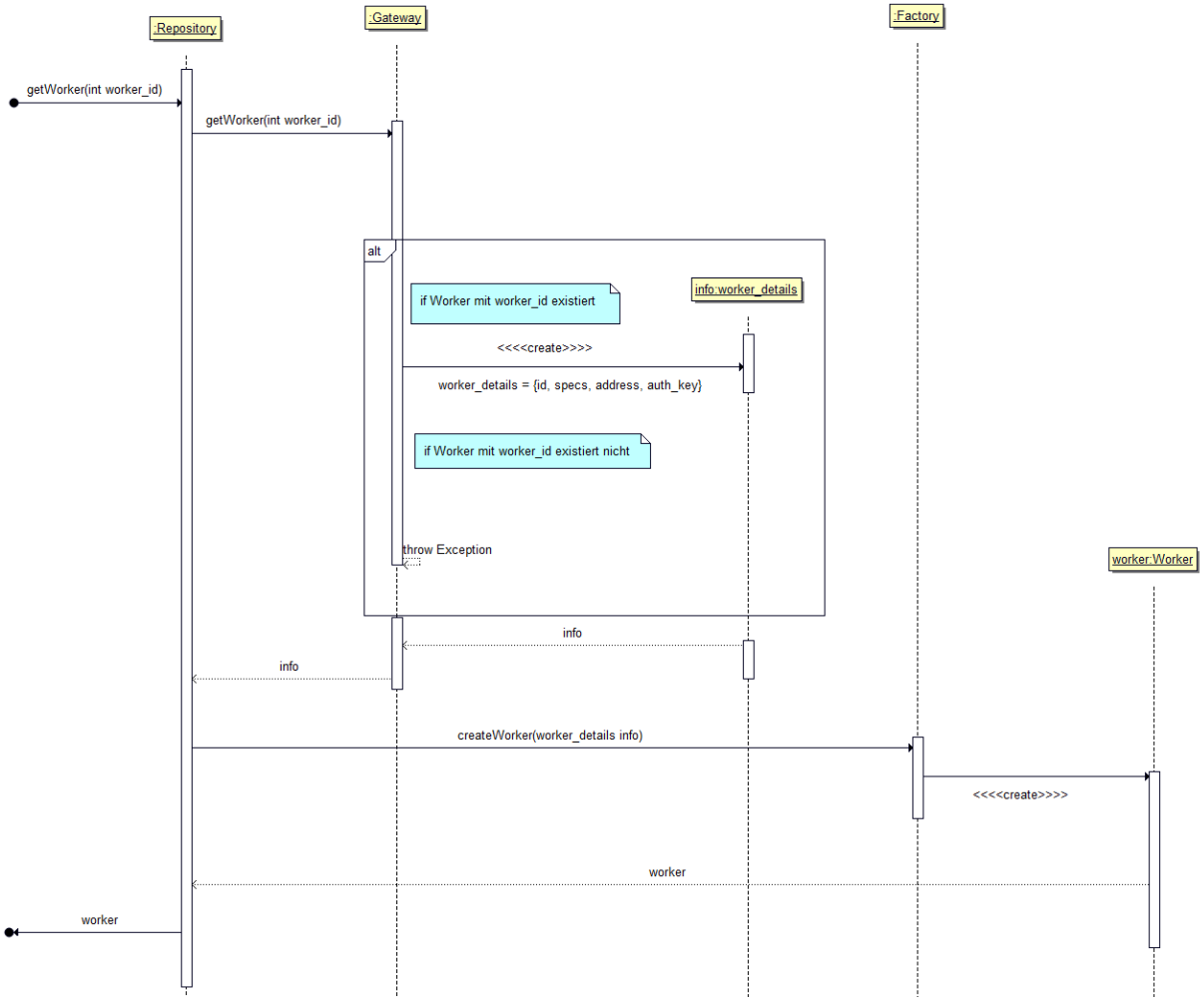
Methoden

- *public void dispatch(Event eventType, std::string worker, int taskID)* Verteilt ein ein Event an Worker. Protokolliert zudem diese Events.

4 Abläufe

4.1 Repository Pattern Beispielverwendung

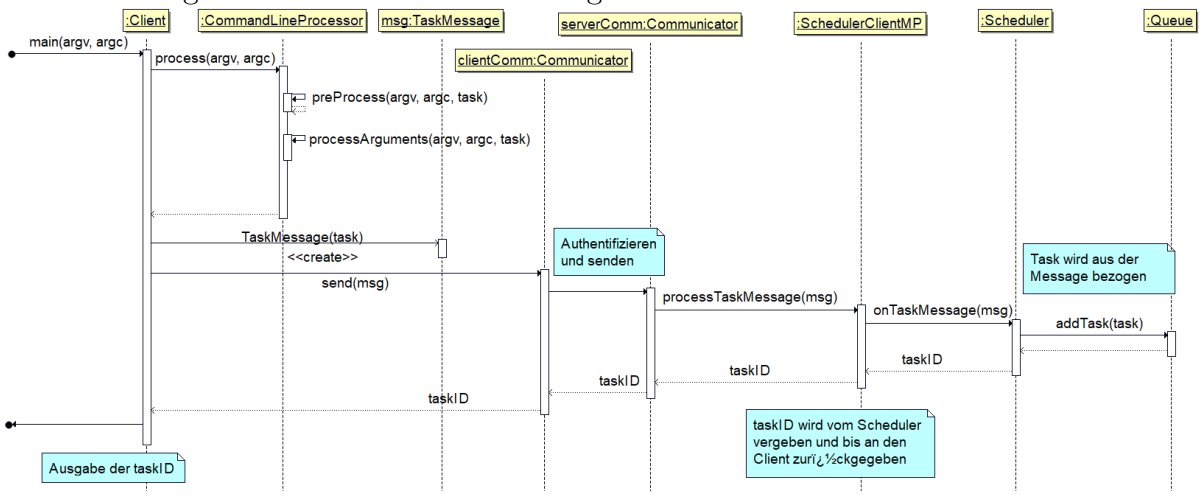
Das folgende Sequenzdiagramm veranschaulicht, wie die *getWorker(uint64_t worker_id)* funktioniert und, was noch wichtiger ist, wie das Repository-Pattern als Ganzes funktioniert.



Der Rest des Programms verwendet die Repository-Klasse als Schnittstelle für Queries. *public Worker getWorker(uint64_t worker_id)* wird aus dem Repository aufgerufen. Das Repository ruft dann die Methode *getWorker* im Gateway mit dem vom Client übergebenen Argument auf. Wenn die Operation erfolgreich war, übergibt das Repository den Rückgabewert vom Gateway an die Methode *createWorker* in der Factory. Das Repository gibt dann das neu erzeugte Worker-Objekt an den Client zurück.

4.2 Eine neue Aufgabe einreichen

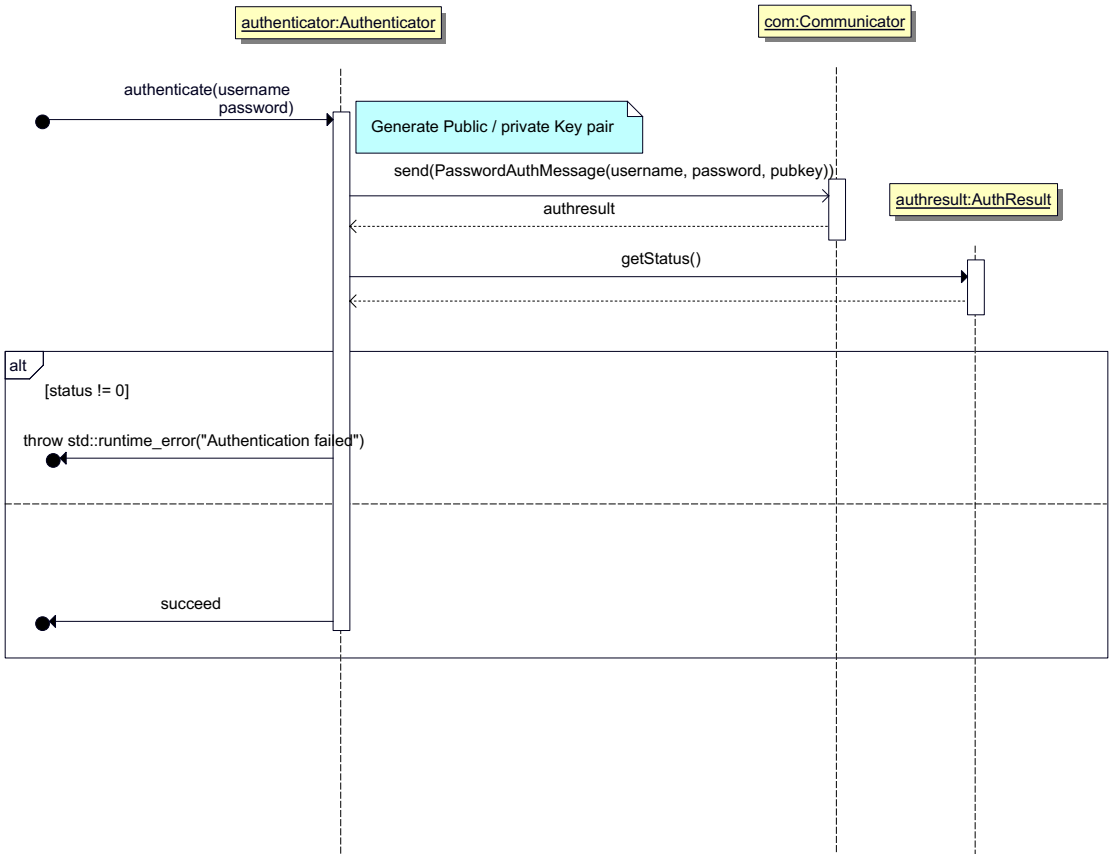
In diesem Sequenzdiagramm ist der Ablauf einer typischen Kommunikation zwischen Client (Benutzer) und Server (Scheduler) veranschaulicht. In diesem Fall gibt der Benutzer eine neue Aufgabe in Auftrag. Andere Anfragen, wie etwa eine Statusabfrage einer vorhandenen Aufgabe finden nach einem analogen Ablauf statt.



Ein anderer Befehl des Benutzers, etwa die Nachfrage, wie der Bearbeitungsstatus einer Aufgabe derzeit ist, folgen diesem Ablauf. Sie unterscheiden sich im wesentlichen nur in den Aktionen, die auf Seiten der Server Anwendung stattfinden, sowie der Rückgabe, die der Server an den Client übermittelt.

4.3 Erstmalige Anmeldenung eines Clients

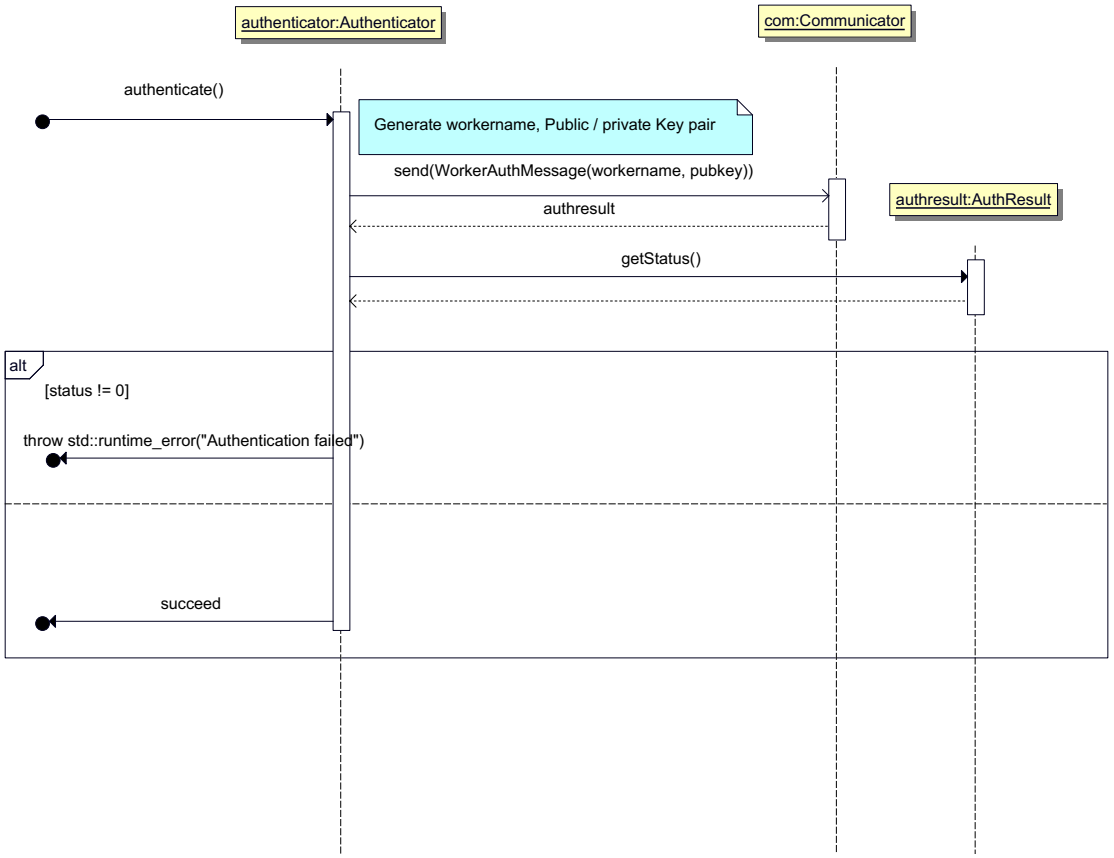
Der Benutzer wird nach seinen Passwort gefragt, da er sich noch nie zufor von diesem Client am Scheduler angemeldet hat. Dafür ruft der Client die authenticate Methode des Authenticators mit Benutzername und Passwort auf. Um das nächste mal ohne Passwort sich authorisieren zu können wird eine public, private Key Schlüsselpaar erzeugt. Anschließend wird der Benutzername, Passwort und publickey an den Scheduler gesendet.



Falls die Authentifizierung erforderlich ist, wird der Benutzername und public Key in der Datenbank des Scheduler gesichert

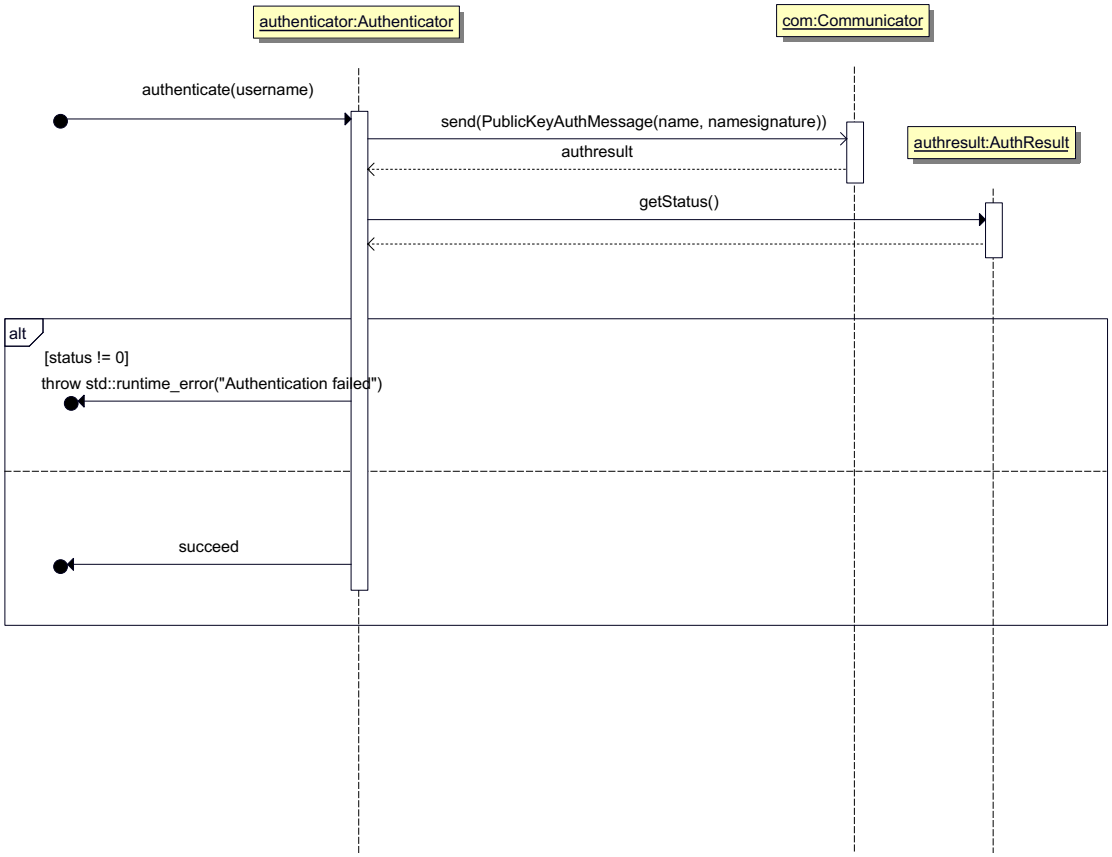
4.4 Erstmalige Anmeldenung eines Workers

Der Worker erzeugt einen eindeutigen Namen und erzeugt ein public, private Key Schlüsselpaar, um wiedererkannt zu werden. Anschließend wird der generierte Name des Worker und der publickey an den Scheduler gesendet.

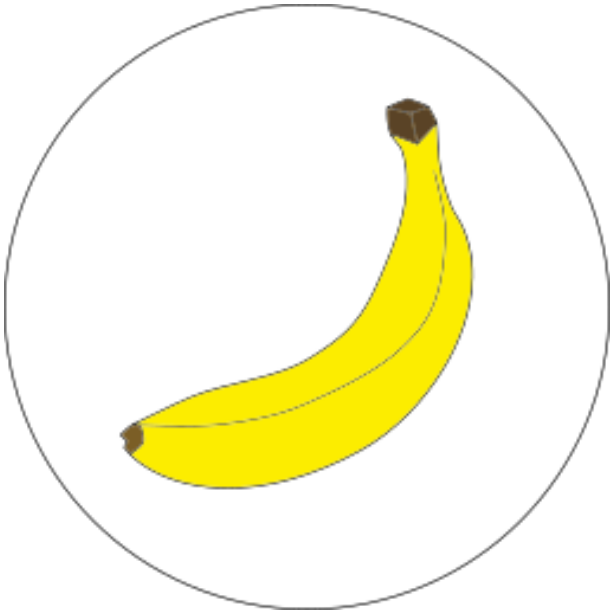


4.5 Passwortlose Anmeldung Client / Worker

Der Client und Worker senden hier statt eines passworts eine signarur, die vom zuvor generierten private key erzeugt wird. Diese wird anschließend vom Scheduler verifiziert.



Ist die Authentifizierung ungültig, wird der Anmeldevorgang mit einer Ausnahme beendet.



Balanced Banana

A Distributed Task Scheduling System

Implementierung

Niklas Lorenz, Thomas Häuselmann, Rakan Zeid Al Masri, Christopher Lukas Homberger und Jonas Seiler

6. März 2020

Inhaltsverzeichnis

1 Einleitung

Dieses Dokument dient dazu, einen allgemeinen Überblick darüber zu geben, was wir, das BalancedBanana-Team, in der Implementierungsphase gemacht haben.

Das erste Abschnitt beschreibt die Probleme, die wir während dieser Phase hatten, und die Änderungen, die wir gegenüber dem ursprünglichen Pflichtenheft und Entwurf vorgenommen haben. Dieser Abschnitt ist in mehrere Unterabschnitte unterteilt, die jeweils ein Modul unseres Programms beschreiben.

Das nächste Abschnitt listet kurz die verschiedenen externen Bibliotheken und Programme auf, die wir verwendet haben.

Danach folgt ein Abschnitt über allgemeine Statistiken über unser Programm einschließlich der Testabdeckung.

Das letzte Abschnitt gibt einen allgemeinen Überblick über unseren ursprünglichen Zeitplan und den tatsächlichen Zeitplan für das Projekt.

2 Probleme, Änderungen am Pflichtenheft und Entwurf

2.1 Allgemein

Viele der Probleme zu Beginn der Phase standen im Zusammenhang mit der Einrichtung unserer einzelnen Systeme. Die Konfiguration der verschiedenen Entwicklungsumgebungen war zu Beginn sehr zeitaufwändig und die Fehlerbehebung war manchmal schwierig, da jede Person eine andere Einrichtung hatte. Es wäre viel einfacher gewesen, wenn wir alle die exakt gleiche Entwicklungsumgebung auf dem gleichen Betriebssystem verwendet hätten. Durch teilweiße fehlender Synchronisierung mit der master Branch, entstanden vermeidbare Merge Konflikte. Wegen vieler Zyklen zwischen den Paketen, wurde die Task klasse in communication verschoben. Jedoch wurden weitere größere Paketänderungen in die nächste Phase verschoben, um die Implementierung und ausstehende Merges nicht verkomplizieren.

2.2 Änderungen am Pflichtenheft

2.2.1 CommandLineInterface

Befehle wurden restrukturiert, benutzen jetzt unterbefehle (wie bei git) Verkürzung der Abkürzungen: -minc -> -c, -maxc -> C, minr -> -r, -maxr -> -R, -ri -> I, -ai -> i, weil CLI11 für Options Abkürzungen einen einzelnen Buchstaben erfordert.

2.2.2 Communication

Automatische Verbindung des Communicators mit dem Schedulers, nur mit gespeicherter Adresse des Schedulers möglich.

2.3 Änderungen am Entwurf

2.3.1 Datenbank

Viele der früheren Probleme mit der Implementierung der Datenbank waren mit dem Setup und meiner Unkenntnis von MySQL verbunden. Im Allgemeinen gab es jedoch keine großen Probleme mit der Implementierung des Datenbankteils unseres Programms. Ich habe die Zeit, die es nicht nur für die Implementierung der Datenbank, sondern auch für das Testen der Datenbank brauchte, grob unterschätzt. Das Testen der Datenbank nahm aufgrund der Natur relationaler Datenbanken enorm viel Zeit in Anspruch. Die aktuelle Implementierung folgt im Allgemeinen dem ursprünglichen Entwurf, es gab jedoch einige Änderungen.

Eine große Änderung war die Aufspaltung der Gateway-Klasse in drei kleinere Klassen (Job-, Worker- und UserGateway). Damit sollte vermieden werden, dass es eine große Götterklasse gibt, die schwer zu warten und zu testen ist. Alle Gateway-Klassen erben von IGateway. Worker- und UserGateway haben sich im Vergleich zum ursprünglichen Entwurf nicht viel geändert, jedoch wurden neue Methoden zu JobGateway hinzugefügt, die von anderen Teilen unseres Programms benötigt wurden.

Eine weitere Änderung war das Hinzufügen eines Caches zum Repository. Der Cache ermöglicht es unserem Programm, verschiedene Objekte zu speichern, so dass andere Teile unseres Programms die Datenbank nicht ständig abfragen müssen.

Eine kleinere Änderung bestand darin, dass die IGateway-Klasse die Verbindung zur Datenbank aufbauen konnte, anstatt sie von der Gateway-Klasse erstellen und ein QSqlDatabase-Objekt als Klassenmitglied speichern zu lassen. Diese Änderung wurde aufgrund einer genaueren Untersuchung der Dokumentation vorgenommen.

2.3.2 Config Files

Da es sich bei den Config Files um sehr einfache Container handelt, die keine große Funktion haben, musste hier nichts groß geändert werden. Lediglich die SchedulerConfig Klasse wurde in ApplicationConfig umbenannt, da sie auch in der Worker-Anwendung eingesetzt werden kann.

2.3.3 Communication

Das Communication Paket war im Entwurf noch sehr abstrakt gehalten. Daher ist während der Implementierung vieles dazugekommen, besonders neue Message-Typen. Mittlerweile gibt es:

- AuthResultMessage
- ClientAuthMessage
- HardwareDetailMessage
- PublicKeyAuthMessage
- RespondToClientMessage
- TaskMessage
- TaskResponseMessage
- WorkerAuthMessage
- WorkerLoadRequestMessage
- WorkerLoadResponseMessage

Eine weitere große Änderung ist die Aufteilung des Communicator in den Communicator und den CommunicatorListener. Letzterer übernimmt jetzt das Warten an einem Port auf Verbindungsanfragen, während der Communicator nur noch für das Empfangen und Senden von Messages verantwortlich ist. Für die Aufteilung haben wir und deshalb entschieden, da auch in allen Netzwerk-Bibliotheken zwischen Socket und Server unterschieden wird. Für das Verarbeiten von Messages sind weiterhin die Unterklassen des MessageProcessor verantwortlich. Diese befinden sich jetzt in dem entsprechenden Anwendungspaket (Scheduler, Worker oder Client) und nicht länger im Communication Paket, da dieses nur als Grundlage für die Kommunikation dienen soll, nicht jedoch die gesamte Verarbeitung für alle drei Anwendungen.

3 Externe Abhängigkeiten

3.1 Im Programm

QT Bibliothek für Netzwerkübertragung und Datenbanken. Wird verwendet, um die drei Programme (Client, Scheduler und Worker) miteinander kommunizieren lassen zu können. Wird außerdem dazu verwendet die MySQL Datenbank auf dem Scheduler zu verwalten und von innerhalb des Codes SQL Anfragen zu ermöglichen.

CLI11 Bibliothek zum einlesen der Befehlszeile. Wird verwendet um die Argumente beim Aufruf der drei Programme (Client, Scheduler und Worker) die der Benutzer angegeben hat einzulesen.

OpenSSL Bibliothek zur Verschlüsselung von Daten. Wird dazu verwendet einen Benutzer mithilfe eines Public-Private-Key Verfahrens mit dem Scheduler zu verbinden und zu identifizieren, wenn sich der selbe Benutzer erneut im System anmeldet.

GTest Bibliothek zum Testen von C und C++ Code. Wird verwendet, um die einzelnen Bestandteile der Programme zu testen.

SimpleHTTPServer Selbstentwickelte Bibliothek, die das Aufsetzten und verwalten eines HTTP Servers ermöglicht. Wird verwendet, um den HTTP Server zu verwalten, der auf HTTP Anfragen bezüglich des Systemstatus (z.B. Auslastung der Arbeiter Rechner, ...) antwortet.

Docker Bibliothek zum kapseln von Prozessen in Containern. Wird dazu verwendet, die Benutzer Aufträge voneinender zu trennen und die gesetzten Grenzen für die Hardwareressourcen einzubehalten.

Cereal Bibliothek zur Serialisierung.

criu Bibliothek zur Checkpoint-/Wiederherstellungsfunktionalität

python Programmiersprache für Scripts. Wird verwendet, um gewisse Tests auszuführen, die Netzwerkfunktionalität erfordern.

3.2 Auf GitHub

CircleCI Ein Continuous-Integrationsprogramm(CI). CI ist eine Softwareentwicklungsstrategie, die die Entwicklungsgeschwindigkeit erhöht und gleichzeitig die Qualität des Codes, den die Teams einsetzen, sicherstellt. Die Entwickler commiten kontinuierlich Code in kleinen Schritten (zumindest täglich oder sogar mehrmals täglich), der dann automatisch erstellt und getestet wird, bevor er mit dem gemeinsamen Repository zusammengeführt wird.

TravisCI Ein weiteres Continuous-Integrationsprogramm.

Coveralls Programm, um die Testabdeckung unseres Codes zu zeigen.

4 Statistik

4.1 Allgemein

Source 4452 LOC

TestSource 8649 LOC

Include 19994 LOC

Insgesamt 53307 LOC (inklusive CMake, SQL, usw.)

4.2 Testfälle

3 Tests von 265 schlagen fehl (master) Coverage 92%

4.2.1 Nicht Abgedeckt

Docker Checkpoint Tests brechen frühzeitig ab und bleiben unberücksichtigt. Unvollständige Einstiegspunkte, werden nicht durch Tests ausgeführt und sind negativ bewertet. Externe Bibliotheks schnittstellen kann man nicht einfach fehlschlagen lassen.

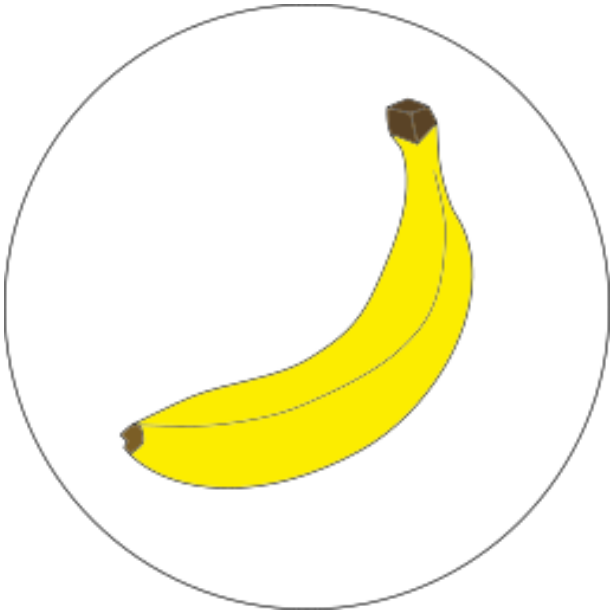
5 Implementierungsplan

5.1 Geplant

	Woche 1	Woche 2	Woche 3	Woche 4	Woche 5?
Authenticator	Christopher				
Queue	Jonas				
MsgProcessor		Niklas			
Datenbank-Schnittstelle	Rakan				
Commandlineinterface	Thomas				
Docker Kernfunktionalität		Christopher			
DB Workerlistener		Jonas			
Communicator	Niklas				
DB SQL		Rakan			
Client, DB Bedienung		Thomas			
Netzwerksocket			Christopher		
Worker Kernfunktionalität			Jonas		
DB HTTP&SMTP			Niklas		
HTTP&SMTP Server			Rakan		
Timer			Thomas		

5.2 Realität

						80% Testabdeckung	
		Woche 1	Woche 2	Woche 3	Woche 4	Woche 5?	Woche 6 (Durch integration gesunkene Testabdeckung)
Angefangen	Authenticator	Christopher			Christopher	Nur der MP ist nicht da	Fertig
Verzug	Queue	Jonas				ok	Fehlerbereinigung ausstehend
Fertig ohne Tests					Get jobids of last n hours		Fertig
Fertig mit Tests					Get Jobids of worker id		Fertig
Integration	MsgProcessor		Niklas	Niklas	Niklas		Alle notwendigen Messages sind / api ist implementiert
	Datenbank-Schnittstelle	Rakan				Höhere Testabdeckung	Anleitung zum aufsetzen fehlt
	Commandlineinterface	Thomas					Fertig
	Docker Kernfunktionalität		Christopher		*Tests implementiert / Fehlerkorrektur*		
	Docker Snapshot					unstabiles docker feature, online Test fails	Checkpoints (experimentell funktioniert nicht garantiert)
	DB Workerlistener	←Arbiter	Jonas		Jonas	Aufgelöst -> Listener	
	Communicator	Niklas					Praxis test ausstehend
	Communicator Listener	Christopher					Praxis test ausstehend
	DB SQL		Rakan		Aufteilen von Repository und Gateway	add credentials to constructor	Fertig
	Client, DB Bedienung		Thomas		Thomas		Fehlerbereinigung ausstehend
	Worker Kernfunktionalität			Jonas	Jonas		Bibliotheken sind schon alle da
	DB HTTP&SMTP			Niklas	Niklas		Fertig (Rakan)
	HTTP&SMTP Server			Christopher	Christopher	Convert Method to class	Testen gegen über dem KIT smtp server fehlt
						needs to get (poll) data	Callbacks über die Repository erstellen
	Timer			Thomas			Mögliche Fehlerbereinigung ausstehend
	Configfiles	Niklas/Christopher					
	DB SQL für Webserver (Callbacks)				Rakan		Callbacks über die Repository erstellen
	Scheduler/worker					Get current worker load	
						constructor	
	Workerstatuschangelistener						Observer
	Factory						2/3 Worker Factory braucht anpassungen
	Task				wie (de)serialisiert man		Fertig
	Worker					get current load / ram ..	Schedulerseitig fertig
	User					constructor/getjobids per user	



Balanced Banana

A Distributed Task Scheduling System

Qualitätssicherung

Niklas Lorenz, Thomas Häuselmann, Rakan Zeid Al Masri, Christopher Lukas Homberger und Jonas Seiler

30. März 2020

Inhaltsverzeichnis

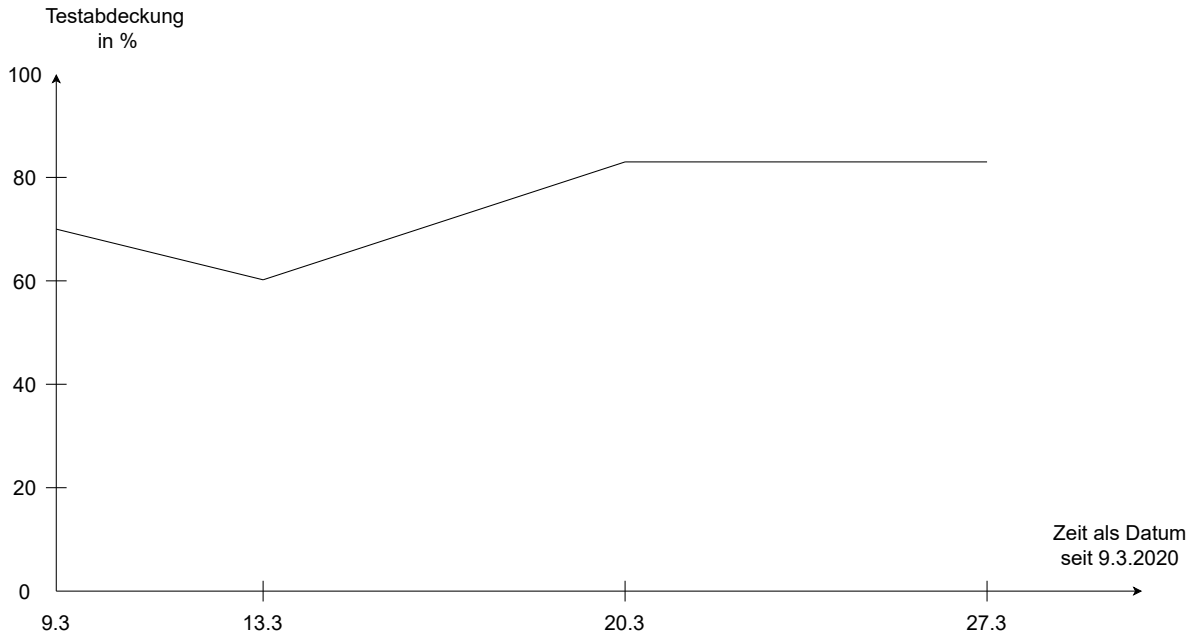
1 Einleitung

Dieses Dokument dient als Übersicht der Qualitätssicherung. Erläutert werden sowohl unsere Methoden zur Qualitätssicherung als auch das Ergebnis dieser. Ebenfalls beinhaltet dieses Dokument eine Anleitung zum Aufsetzen und Benutzen des endgültigen Programms.

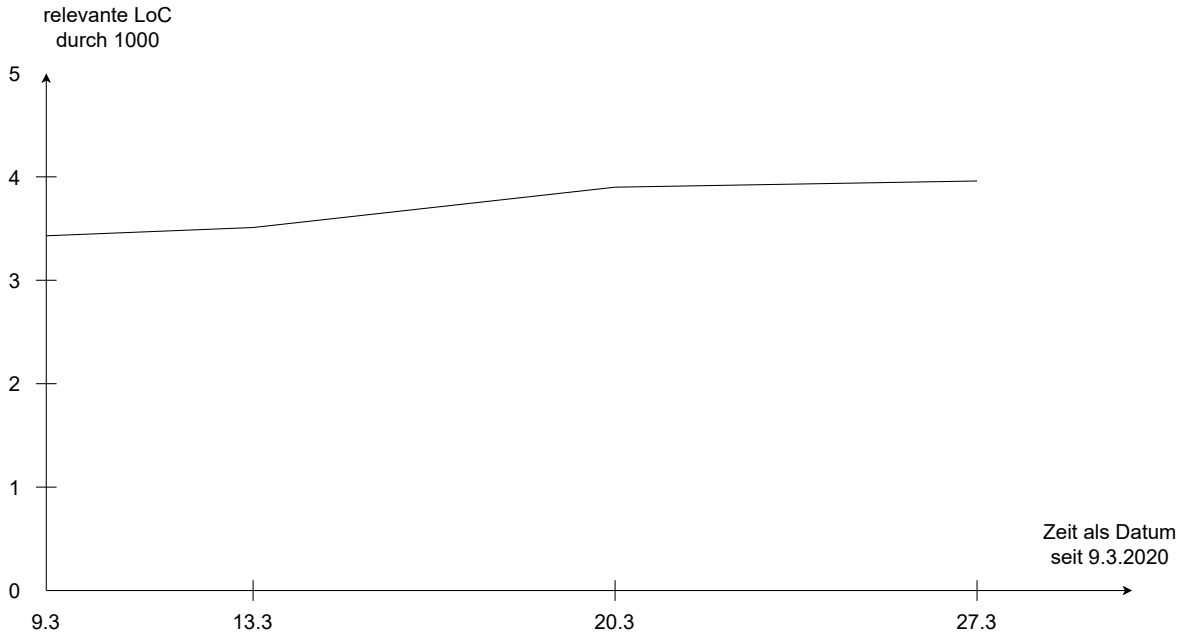
2 Testabdeckung

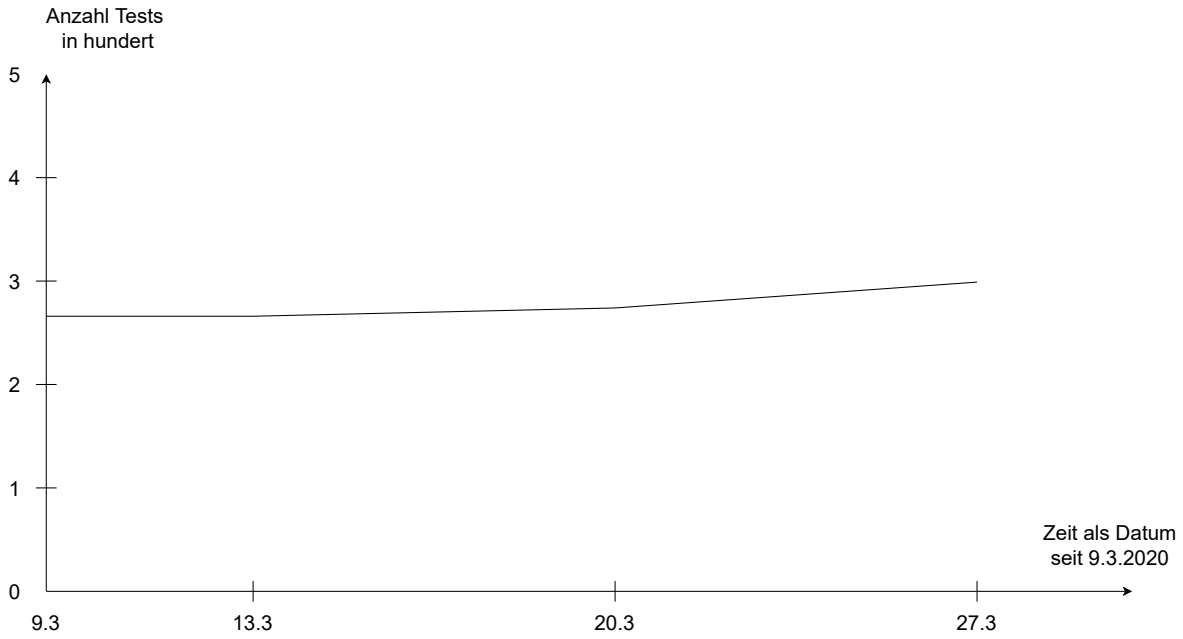
2.1 Klassenüberdeckung

Die Testüberdeckung der einzelnen Klassen ist in der ersten Woche der Qualitätssicherung gefallen, dies liegt daran, dass einige Klassen zwar intern funktioneren, sie jedoch im Zusammenspiel mit anderen nicht. In der ersten Woche wurden also hauptsächlich fehlerhafte Schnittstellen oder andere Fehler beseitigt und entsprechende Unit Tests erst in den folgenden Wochen eingeführt.



Entsprechend sehen wir einen Anstieg in Lines of Code (LOC) sowie der Anzahl Tests.





Da der Großteil des Codes bereits implementiert ist, sowie Integrationstests eher länger dafür aber weniger als Unittests sind, steigt auch die Anzahl der Tests nicht stark.

2.2 Integrationstests

Für das Automatisierte Testen des ganzen Systems haben wir ein Skript in Python benutzt, dieses beinhaltete diverse Szenarien und prüfte mit Positiv- als auch mit Negativtests. Neben diesem haben wir auch einiges "von Hand"getestet. Die, im Pflichtenheft definierten, Testfälle werden wie folgt abgedeckt:

T	Kurzbeschreibung	Wie getestet?
T1	Verbinden des Clients mit Server	Automatisch
T2	Festlegen von Prioritäten	Manuell
T3	Festlegen des Betriebssystems	Automatisch
T4	Abfragen eines Aufgabenstatus	Manuell
T5	Benachrichtigung bei Abschluss einer Aufgaben	Manuell
T6	Erstellen von Sicherungen	-
T7	Anforderung von Ausgabe	Manuell
T8	Abbrechen von zu langen Aufgaben	-
T9	Manuelles Stoppen von Aufgaben	Manuell
T10	Manuelle Sicherung von Aufgaben	Manuell

In den Integrationstests haben wir hauptsächlich Verhalten getestet die so im Pflichtenheft noch nicht voraussagbar waren, daher haben wir tatsächlich mehr Tests als nur durch die vordefinierten ersichtlich.

3 Umgesetzte Funktionale Anforderungen

FA	Kurzbeschreibung	Implementiert?	Überdeckt von?
FA1	Client verbindet sich mit Server	Ja	T1
FA2	Benutzer authentifiziert sich	Ja	Integrationstest
FA3	Benutzer kann Aufgaben einreihen	Ja	T2 & T3
FA4	Benutzer kann Parameter übergeben	Ja	T2 & T3
FA41	Client speichert Parameter in Config	Ja	Manuelltest
FA42	Benutzer kann Config übergeben	Ja	Manuelltest
FA43	Benutzer kann Priorität festlegen	Ja	T2
FA44	Benutzer kann min und max Cores festlegen	Ja	Manuelltest
FA45	Benutzer kann min und max RAM festlegen	Ja	Manuelltest
FA46	Benutzer kann Betriebssystem festlegen	Ja	T3
FA47	Benutzer kann angeben ob Client blockiert	Ja	Manuelltest
FA48	Benutzer übergibt Pfad zu Aufgabe	Ja	Integrationstest
FA49	Es können Standardwerte benutzt werden	Ja	Integrationstest
FA5	Benutzer kann Status von Aufgabe einsehen	Ja	T4
FA6	Benutzer bekommt Email bei Abschluss	Ja	T5
FA7	Server erstellt Backups	(Nein)	T6
FA8	Benutzer kann Ausgabe anfordern	Ja	T7
FA9	Benutzer kann Statistiken abfragen	Ja	Manuell
OFA1	Benutzer kann Restzeit einsehen	Nein	-
OFA2	Server stoppt lange Aufgaben	Nein	T8
OFA3	Benutzer kann manuell Aufgaben stoppen	Nein	T9
OFA4	Benutzer kann manuell Aufgaben sichern	(Nein)	T10
OFA5	Benutzer kann Pausierbarkeit angeben	(Nein)	-

Für Anforderungen mit einem (Nein) existiert zwar in manchen Teilen des Programms die Funktionalität dafür, die Funktion selber ist jedoch nicht umgesetzt oder nutzbar.

4 Benutzeranleitung

4.1 Aufsetzen des Schedulers

Zuerst müssen wir das System aufsetzen, das Programm wird in einem Paket geliefert, die entsprechende Funktionen finden sich dann in den einzelnen ausführbaren Dateien.

- Herunterladen des Programms unter
  `https://github.com/balancedbanana/balancedbanana/releases`
- Einrichten der MySql Datenbank mit

  ```
  cat balancedbanana.sql | sudo mysql
  ```

  Die SQL Datei ist im Archiv unter
  `share/balancedbanana/balancedbanana.sql`
  zu finden.
  Nun ist die Datenbank unter dem Schema balancedbanana installiert. In der MySQL-Konsole einen Datenbank Nutzer für balancedbanana anlegen mit folgenden MySQL-Befehlen:

  ```
  CREATE USER 'balancedbanana'@'localhost' IDENTIFIED BY
  'balancedbanana';

  GRANT ALL PRIVILEGES ON balancedbanana.* TO 'balancedbanana'@'localhost';

  FLUSH PRIVILEGES;

  exit
  ```

- Configfiles pro Benutzer anlegen unter
  `$HOME/.bbc` bzw `.bbs` oder `.bbd`.
  Dazu einfach die Configfiles von
  `share/balancedbanana/.bbc` bzw `.bbs` oder `.bbd`
  kopieren.
- Das Passwort `balancedbanana` nach `IDENTIFIED BY` bei Bedarf anpassen und in
  `$HOME/.bbs/appconfig.ini`
  in Zeile `databasepassword` anpassen.

Nun ist der Scheduler aufgesetzt.

4.2 Benutzen des Clients

Der Client kann mit dem Programm `bbc` gestartet werden. Der Client hat verschiedene Subcommands sowie Flags für diese, nachführend sind diese erläutert. Für nicht angegebene Parameter wird zuerst in der Benutzerconfig (`$HOME/.bbc`) dann in der Applikationsconfig (`share/balancedbanana/.bbc`) und danach wird entweder auf Standardwerte oder auf gesetzte Standardparameter im Scheduler gesetzt. Bei Unklarheiten der Kommandos oder deren Parameter kann immer mit dem Argument -h ein Hilfemenü aufgerufen werden.

4.2.1 Run

Der Run-Command reiht eine Aufgabe ein, folgende Parameter sind verfübar:

- –server. Gibt die IP Adresse des Schedulers an. Angaben als String.
- –port. Gibt den Port am Scheduler an. Angaben als 16 Bit Integer.
- –block / -b. Gibt an ob der Client blockiert, bis die Aufgabe beendet ist. Angaben als Boolean.
- –email / -e. Gibt die Email an, die benachrichtigt wird, wenn die Aufgabe beendet ist. Angaben als String.
- –image / -i. Gibt den Pfad des Images an, dass für die Aufgabe benutzt werden soll. Angaben als String.
- –priority / -p. Gibt die Priorität der Aufgabe an. Mögliche Werte sind 'low','normal', 'high' und 'emergency'.
- –max-cpu-count / -C. Gibt die maximale Anzahl, der benutzten Kerne, der Aufgabe, an. Werte als 32 Bit Integer.
- –min-cpu-count / -c. Gibt die minimale Anzahl, der benutzen Kerne, der Aufgabe, an. Werte als 32 Bit Integer.
- –max-ram / -R. Gibt den maximal nutzbaren Arbeitsspeicher in miB an. Werte als 64 Bit Integer.
- –min-ram / -r. Gibt den minimal nutzbaren Arbeitsspeicher in miB an. Werte als 64 Bit Integer.
- –job / -j. Gibt die Eingabe an, die den Job startet. Alles hinter dieser Eingabe wird als Kommando aufgefasst, dieses Argument sollte also das letzte des Run Kommandos sein. Angaben als String.

Beispiel:

```
./bbc run --server 192.168.178.82 --port 25565 -b true -e example@hello.com
-i catsimulation -p high -C 4 -c 1 -R 4096 -r 1024 -j sh catsimulationfile.sh
```

4.2.2 addImage

Der addImage-Command registriert ein Image, sodass es mit –image eingereiht werden kann. Das einzige Argument ist der Pfad zum Dockerfile. Beispiel:

```
./bbc addImage catsimulation C:/Users/Jonas/Simulations/Catsimulation
```

4.2.3 removeImage

Entfernt ein registriertes Image unter dem gegebenen Namen. Das einzige Argument ist der Name des Images. Beispiel:

```
./bbc removeImage parrotsimulation
```

4.2.4 status

Frägt den Scheduler nach dem Status eines Jobs. Das einzige Argument ist die ID des Jobs als unsigned Integer. Beispiel:

```
./bbc status 42
```

4.2.5 tail

Frägt den Scheduler nach den letzten ausgegebenen Zeilen der Aufgabe. Das einzige Argument ist die ID des Jobs als unsigned Integer. Beispiel:

```
./bbc tail 42
```

4.2.6 stop

Stopt den angegebenen Job. Das einzige Argument ist die ID des Jobs als unsigned Integer. Beispiel:

```
./bbc stop 43
```

4.2.7 pause

Pausiert den angegebenen Job. Das einzige Argument ist die ID des Jobs als unsigned Integer. Beispiel:

```
./bbc pause 43
```

4.2.8 continue

Setzt den angegebenen Job fort. Das einzige Argument ist die ID des Jobs als unsigned Integer. Beispiel:

```
./bbc continue 43
```

4.2.9 backup

Sichert den angegebenen Job. Das einzige Argument ist die ID des Jobs. Beispiel:

```
./bbc backup 43
```

4.2.10 restore

Stellt den angegeben Job aus einem Backup wieder her. Als Argument erhält es die Job ID und die Backup ID des Jobs. Beispiel:

```
./bbc restore 43 240
```

4.3 Benutzen des Schedulers

Der Scheduler hat im Gegensatz zum Client keine unterschiedlichen Kommandos und muss nur gestartet werden. Bei Unklarheiten kann wieder mit -h ein Hilfemenü aufgerufen werden. Folgende Parameter stehen zur Verfügung:

- –server / -s. Legt fest auf welcher IP-Adresse der Scheduler gestartet wird. Angaben als String.
- –webapi / -w. Legt fest auf welcher IP-Adresse die WebAPI des Schedulers gestartet wird. Angaben als String.
- –serverport / -S. Legt fest auf welchem Port der Scheduler gestartet wird. Angaben als 32 Bit Integer.
- –webapi-port / -W. Legt fest auf welchem Port die WebAPI gestartet werden soll. Angaben als 32 Bit Integer

Hier ein Beispielaufruf:

```
./bbs -s 192.168.178.81 -w 192.168.178.82 -S 25565 -w 25566
```

4.4 Benutzen des Workers

Der Worker hat ebenfalls nur ein Kommando womit er gestartet wird. Bei Unklarheiten kann ebenfalls wieder mit -h ein Hilfemenü aufgerufen werden. Folgende Parameter stehen zur Verfügung:

- –server / -s. Legt fest welche IP-Adresse der Scheduler hat. Angaben als String.
- –serverport / -S. Legt fest auf welchem Port der Scheduler gestartet wurde. Angaben als 32 Bit Integer.

Hier ein Beispielaufruf:

```
./bbd -s 192.168.178.82 -S 25565
```

4.5 DevSetup

Die Installationsanleitung für Entwickler, einschließlich einer Liste aller Pakete, Frameworks und Bibliotheken, sind unter
`https://github.com/balancedbanana/balancedbanana/blob/master/README.md` zu finden.

5 Probleme und Bugs

Die Phase der Qualitätssicherung wurde durch eine gewisse Pandemie gestört, daher fehlt uns eine richtige Testumgebung. Das Testen unter realen Umständen kam deshalb zu kurz oder gar nicht vor. Wir wissen aktuell noch nicht gut wie sich das Programm mit mehreren Workern verhält.

Ebenfalls haben wir zwar die Benachrichtigungsmails privat getestet, wissen jedoch nicht, ob diese fehlerfrei mit dem KIT-Mailserver funktionieren.

Die Anleitung zum Aufsetzen und Benutzen des Systems funktioniert auf unseren Systemen zwar, jedoch wissen wir nicht ob es auf den Systemen des CES zu anderweitigen Problemen kommen kann.

Wir schätzen bei diesen Realtests noch einige Bugs zu finden, dies können wir jedoch aktuell nicht testen.

Einige Funktionen, wie das Stoppen und Fortführen von Jobs, sind, wegen des experimentellen Checkpoint Features von Docker, in keinem konsistent ausführbaren Zustand. Das Aufsetzen der Funktionen ist kompliziert und deshalb haben wir die Funktion vorerst als implementiert aber nicht nutzbar angemerkt.

Ein kleineres Problem ist das Beschreiben des RAMs. Beim Scheduler besteht die Chance, dass sich die Daten im Arbeitsspeicher festsetzen und diesen somit irgendwann vollschreiben. Wir empfehlen also, den Scheduler einmal alle paar Monate neu zu starten.

6 Konklusion

Da die Qualitätssischerungsphase mit der Quarantäne kollidierte, fehlt uns leider Erfahrung mit der echten Benutzung des Systems. Trotzdem schafft unser Programm die von uns gestellten Szenarien. Viele optionale Features existieren ebenfalls als Funktion, wir empfehlen jedoch nicht diese aktuell zu benutzen. Wir denken, dass unser Programm an einem Punkt angekommen ist, an dem es mit realem Testen, und sukzessiven Beseitigen der Probleme, gut funktionieren sollte. Zusammenfassend sind wir zufrieden mit dem Ergebnis.