Goethe-Universität Frankfurt am Main

Ein verteiltes Experiment-Control-System für das Compressed-Baryonic-Matter-Experiment

Masterarbeit

Daniel Bruins
Matrikel-Nummer 6204842

Erstprüfer Prof. Dr. Volker Lindenstruth
Zweitprüfer Prof. Dr. Udo Kebschull

Erklärung gemäß Master-Ordnung Informatik 2015 § 35 Abs. 16

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Ebenso bestätige ich, dass diese Arbeit nicht, auch nicht auszugsweise, für eine andere Prüfung oder Studienleistung verwendet wurde.

Ort, Datum Unterschrift

Ziel dieser Arbeit ist die Konzeption und Implementierung eines *Experiment Control Systems* (ECS) für das *Compressed Baryonic Matter* (CBM) Experiment an der zukünftigen FAIR Anlage am GSI Helmholtzzentrums für Schwerionenforschung in Darmstadt.
Im CBM-Experiment kommen eine Vielzahl von unabhängigen Systemen – wie zum Beispiel Detektoren – zum Einsatz. Aufgabe des ECS ist es, diese in einem globalen System zusammenzufassen.
Einige Konzepte des bereits bestehenden ECS für das ALICE-Experiment können auch für das CBM-ECS verwendet werden. Das CBM-Experiment ist im Gegensatz zum ALICE-Experiment selbstgetriggert, wodurch eine größere Unabhängigkeit zwischen den Detektoren entsteht. Diese Unabhängigkeit wird im ECS des CBM-Experiments genutzt, indem im Fehlerfall von einem Subsystem andere Subsysteme unbeeinträchtigt bleiben können.
Ein Konzept, welches von ALICE übernommen werden kann, ist das aufteilen von Detektoren in unabhängige Partitionen, was sich unter anderem zum Testen von Detektoren bewährt hat.
Zustände einzelner Subsysteme werden vom CBM-ECS in *Finite State Machines* festgehalten. Eine wichtige Aufgabe des ECS ist es, aus diesen Zuständen einen Gesamtzustand für Partitionen herzuleiten.
Für das ECS wurde eine hierarchische Struktur aus unabhängigen Komponenten gewählt, welche als eigenständige Prozesse in einem Netzwerk verteilt laufen, und Nachrichten beziehungsweise Befehle untereinander austauschen können.
Für das ECS wird zusätzlich eine webbasierte Nutzeroberfläche konzipiert und implementiert, über welche Benutzer den Zustand des Systems betrachteten können und das Experiment gesteuert wird.
Als Ergebnis der Arbeit ist sowohl ein Konzept für das ECS als auch ein lauffähiger ECS-Prototyp entstanden, mit welchem man Datennahmen für CBM starten kann. Dieser Prototyp kann als Grundlage für ein zukünftiges ECS verwendet werden.

The objective of this master's thesis is the design and implementation of an experiment control system (ECS) for the compressed baryonic matter (CBM) experiment at the future FAIR facility at the GSI Helmholtzzentrum für Schwerionenforschung in Darmstadt. In the CBM experiment multiple independent systems – like for instance detectors – come to use. The task of the ECS is to unite these systems in a global control entity.
Some concepts of the already existing ECS of the ALICE experiment can also be used in the ECS for the CBM experiment. Unlike the ALICE experiment, the CBM experiment is self-triggered, which provides a greater independence between detectors. This independence shall be used in the ECS of the CBM experiment, to achieve that in case of a subsystem error other subsystems remain unaffected.
A concept of the ALICE ECS, which can also be used for CBM, is the partitioning of detectors into several independent partitions, which has proven to be useful for testing detectors.
States of subsystems are being represented in finite state machines by the CBM ECS. An important task of the ECS is to determine a global state for a partition from the states of its subsystems.
For the ECS a hierarchical structure of independent components has been chosen. These components run as independent processes distributed in a network and exchange messages or commands among each other.
In addition a web based user interface has been designed and implemented, through which users can observe the state of the system and control the experiment.
As a result this master's thesis provides a concept as well as a working prototype for the ECS, which is able to start a data taking for CBM. This prototype can be used as the foundation for the future ECS.

Danksagung

Ich möchte mich bei Herrn Prof. Dr. Lindenstruth für die Vergabe dieses interessanten Themas mit vielen Freiräumen bedanken. Außerdem bedanke ich mich bei seiner Arbeitsgruppe für die angenehme Arbeitsatmosphäre. Besonders möchte ich mich bei Herrn Dr. Jan de Cuveland und Dirk Hutter bedanken, die mich während der Arbeit mit Ratschlägen und Anmerkungen unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

In dieser Arbeit geht es um die Konzeption und Implementierung eines *Experiment Control Systems* (ECS), für das Teilchenbeschleuniger-Experiment *Compressed Baryonic Matter* (CBM). Das ECS muss als zentrale Kontrollinstanz über das gesamte Experiment dienen und eine ansonsten unabhängige Vielzahl von Subsystemen zu einer großen Gesamtinstanz verbinden. Außerdem muss es aus den Zuständen der Subsysteme einen Zustand für das gesamte Experiment bestimmen. Für andere Teilchenbeschleuniger-Experimente existieren bereits solche ECS. Die Anforderungen des CBM-Experiments an das ECS unterscheiden sich allerdings von diesen bereits existierenden. Im Folgenden dieses Kapitels wird zunächst das CBM-Experiment erläutert. Danach wird das ECS des ALICE-Experiments erläutert und zum Vergleich herangezogen.

1.1 Das CBM-Experiment

Das *Compressed Baryonic Matter* Experiment (CBM) ist ein Teilchenbeschleuniger-Experiment, welches an der zukünftigen FAIR-Anlage am Standort des GSI Helmholtzzentrums für Schwerionenforschung in Darmstadt stattfinden soll. Mit dem CBM-Experiment sollen innerhalb des QCD-Phasendiagramms Zustände von subatomarer Materie untersucht werden, die unter hohen Baryonen-Dichten existieren.[7]
Abbildung 1.1 zeigt ein solches Phasendiagramm. Interessant sind hierbei die Übergänge vom Hadronen-Gas in das Quark-Gluon-Plasma. In einem

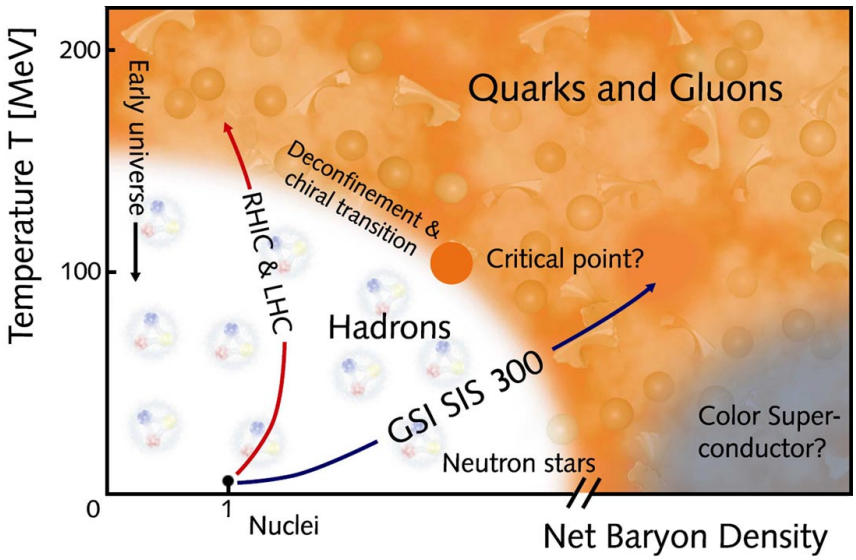

Abbildung 1.1: Das QCD Phasendiagramm. Der blaue Pfeil markiert den Bereich, welcher im CBM-Experiment untersucht werden soll.[7]

Quark-Gluon-Plasma können Quarks und Gluonen sich frei bewegen, während sie ansonsten nur in Verbindung mit anderen Teilchen existieren. Das Quark-Gluon-Plasma kann auf zwei verschiedene Arten erzeugt werden, zum einen indem die Materie erhitzt wird, zum anderem indem die Materie verdichtet wird. Man vermutet, dass dieses Plasma während der Entstehung des Universums mit hoher Temperatur und geringer Dichte existiert hat und innerhalb von Neutronensternen mit geringer Temperatur und hoher Dichte existiert. [13].
Im ALICE-Experiment am CERN werden Zustände mit kleinen Baryonen-Dichten und hohen Temperaturen mit dem LHC-Beschleuniger untersucht, was in Abbildung 1.1 mit dem roten Pfeil eingezeichnet ist. Im Gegensatz dazu sollen im CBM-Experiment Zustände mit geringer Temperatur und hoher Dichte mit dem GSI-SIS-300-Beschleuniger untersucht werden, was von dem blauen Pfeil repräsentiert wird. Durch das CBM-Experiment erhofft man sich unter anderem neue Materien-Zustände im Phasendiagramm zu finden, mehr über die Übergänge ins Quark-Gluonen-Plasma zu erfahren und Kenntnisse über das Innere von Neutronensternen zu gewinnen. [7]
Die Daten in CBM werden durch eine Kollision von Schwerionen mit einem Fixed-Target erzeugt, während zum Beispiel in ALICE Schwerionen miteinan-

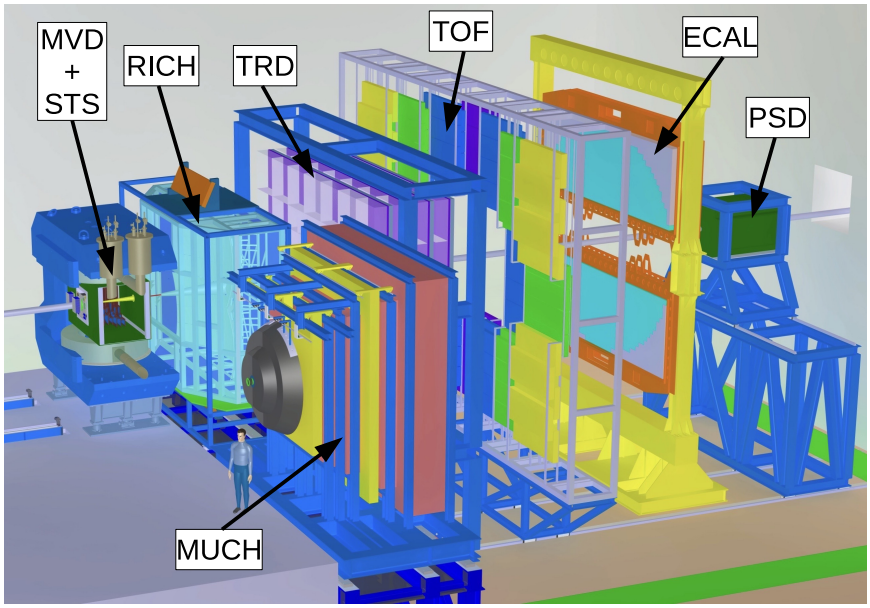

Abbildung 1.2: Der Aufbau des CBM-Experiments. Der Strahl verläuft von links nach rechts durch die einzelnen Detektor Stationen. Der RICH und MUCH Detektor können je nach Konfiguration gegeneinander getauscht werden.[2]

der zur Kollision gebracht werden.
In Abbildung 1.2 sieht man den Aufbau des Experiments mit seinen verschiedenen Detektoren, wobei der Partikelstrahl im Bild von links nach rechts verläuft. Es gibt eine Konfiguration zur Messung von Elektronen und eine zur Messung von Myonen. In der Elektronen-Konfiguration tritt der Strahl zunächst in den *Micro Vertex Detector* (MVD) ein. Nach der Kollision mit dem Target können bei den entstandenen Teilchen Sekundärzerfälle auftreten. Die Aufgabe des MVD ist es, diese Zerfälle und deren Position festzustellen. Danach kommt das *Silicon Tracking System* (STS), in welchem die Teilchenspuren rekonstruiert werden und der Impuls der Teilchen bestimmt wird. Dann folgt der *Ring Imaging Cherenkov Detector* (RICH) zur Identifizierung und Unterdrückung von Pionen. Anschließend kommt der *Transition Radiation Detector* (TRD), welcher Elektronen und Pionen identifiziert. Daraufhin folgt die *Time of Flight Wall* (TOF) zur Erkennung von Hadronen. Als Vorletztes kommt das *Electromagnetic Calorimeter* (ECAL) zur Messung von Photonen. Am Ende des Detektor-Aufbaus befindet sich der *Projectile Spectator Detector* (PSD), welcher die Zentralität und die Ebene der Kollision misst. Im Falle der

Myonen-Konfiguration wird das ECAL entfernt und der RICH-Detektor durch das *Muon Chamber System* (MUCH) zur Erkennung von Myonen ersetzt.[3]
Nach einer Kollision werden im CBM-Experiment etwa 700 Spuren von geladenen Teilchen erwartet, welche von den Detektoren gemessen werden. Ziel der Messung ist es, Spuren zu finden, welche auf besondere Zerfallsereignisse hindeuten. Da die erzeugte Datenmenge während des Experiments zu groß ist um sie zu speichern, muss das Erkennen von interessanten Spuren, welche dann persistent gespeichert werden sollen, zur Laufzeit des Experiments geschehen. Zur Erkennung von interessanten Daten werden in CBM sehr hohe Messraten von bis zu 10 MHz benötigt. Im ALICE-Experiment existiert ein Trigger-System, welches den Detektoren mitteilt, wann sie Daten nehmen sollen. In CBM sind solche Trigger-Systeme nicht möglich, da die notwendige Datenanalyse zu komplex ist, um eine schnelle Trigger-Entscheidung zu treffen. Daher werden alle gemessenen Daten zusammen mit einem Zeitstempel aufgenommen und an den *First Level Event Selector* (FLES) weitergegeben. Der FLES ist neben den Detektoren ein weiteres Subsystem des Experiments und hat dann die Aufgabe, mithilfe der Zeitstempel die Spuren zu rekonstruieren und unter diesen besondere zu finden.[8]
Ein weiteres Subsystem ist das *Timing and Fast Control* (TFC). Die Aufgabe des TFC ist es, dem Auslesen des Experiments einen synchronen globalen Takt und Zeitinformation bereitzustellen. Außerdem verteilt es Steuerungsinformationen in kurzer und konstanter Zeit an die Subsysteme des Experiments.[12]
Weiterhin existiert das *Detector Control System* (DCS), welches die Detektoren in Hinsicht auf physikalische Werte wie Temperatur und elektrische Spannung überwacht, und *Quality Assessment* (QA), welches genommene Daten auf ihre Qualität prüft.

1.2 Das ALICE-*Experiment Control System*

Für andere Experimente existieren bereits *Experiment Control Systems* (ECS). Eines dieser ECS ist das ECS für das ALICE-Experiment, welches in diesem Abschnitt zum Vergleich mit dem benötigten ECS für CBM genauer erläutert

wird.
Das ECS von ALICE dient als zentrale Steuerinstanz des Experimentes. Es hat die Aufgabe, die Detektoren und mehrere ansonsten unabhängige Subsysteme, welche mit den Detektoren interagieren, innerhalb des Experimentes untereinander zu koordinieren. In ALICE sind diese Subsysteme der Trigger, das *Detector Control System* (DCS), das *Data Acquisition System* (DAQ) und der *High Level Trigger* (HLT). Das ECS bietet eine Perspektive, welche alle diese Systeme beinhaltet.[6]
Das ALICE-ECS erlaubt es, die Detektoren in mehrere Partitionen aufzuteilen, sodass diese unabhängig voneinander bedient werden können. Jede Partition enthält zwei Listen. Die erste Liste enthält alle Detektoren, welche der Partition zugewiesen sind und die zweite Liste alle Detektoren, welche aktuell nicht in der Partition aktiv sind. Detektoren können über die erste Liste mehreren Partitionen gleichzeitig zugewiesen sein, aber immer nur in einer Partition aktiv sein. Es ist auch möglich, Detektoren in einer „standalone“-Partition zu betreiben, in der ein Detektor allein betrieben wird, um diesen beispielsweise zu testen. Das ALICE-ECS unterscheidet zwischen globalen Operationen und individuellen Operationen. Globale Operationen werden auf allen Detektoren einer Partition ausgeführt. Die Befehle werden in diesem Fall über den *Trigger Partition Agent* verschickt. Individuelle Operationen werden hingegen auf nur einem Detektor ausgeführt. Hierbei werden Befehle an die *Local Trigger Unit* des Detektors versendet.[4]
Das ECS erhält Status-Informationen von Detektoren und Subsystemen und kann Befehle an diese verschicken. Das Verschicken von Befehlen läuft über Interfaces welche auf *Finite State Machines* basieren. Subsysteme können sowohl direkt über ein eigenes Steuersystem angesprochen werden oder über das ECS gesteuert werden.[4]
Detektoren werden jeweils von einem *Detector Control Agent* (DCA) kontrolliert. Partitionen werden jeweils von einem *Partition Control Agent* (PCA) kontrolliert. Der PCA kann über ein *User Interface* gesteuert werden. Der DCA kann entweder vom PCA kontrolliert werden oder ebenfalls direkt über ein eigenes *User Interface*, falls er im Standalone-Betrieb läuft. Für jedes Interface wird ein eigener Prozess erzeugt. Für jedes System können jeweils viele Benutzer jeweils eine betrachtende Rolle einnehmen, aber nur ein Benutzer eine

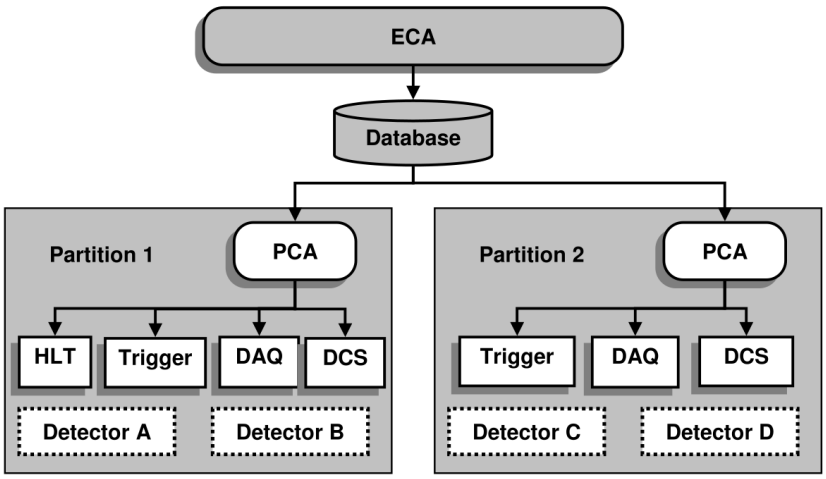

Abbildung 1.3: Beispielhafter Aufbau des ALICE-ECS mit zwei Partitionen und vier Detektoren.[6]

kontrollierende Rolle.[4]
In Abbildung 1.3 sieht man einen beispielhaften Aufbau des ECS von ALICE mit zwei Partitionen und vier Detektoren. Über den Partitionen steht der *Experiment Control Agent* (ECA). Die Aufgabe des ECA ist es, die Datenbank des Systems zu verwalten. In der Datenbank werden alle Ressourcen mit ihren Eigenschaften und die Zuweisungen von Detektoren zu Partitionen abgelegt. Änderungen an Partitionen können nicht während einer Datennahme unternommen werden.[6]
Zur Veröffentlichung von Daten und als Schnittstelle zwischen verschiedenen Komponenten verwendet ALICE-ECS das *Distributed Information Management System* (DIM) [5]. Das DIM wurde ursprünglich für das DELPHI-Experiment am CERN entwickelt. Es dient zur Übertragung und Veröffentlichung von Daten in einem verteilten System. DIM basiert auf dem Client-Server-Modell und erlaubt es, Informationen als Services anzubieten, welche von mehreren Clients abgerufen werden können. Abbildung 1.4 zeigt den Ablauf einer Datenveröffentlichung. Ein Server, welcher Daten veröffentlichen will, registriert einen Service auf einem globalen Nameserver. Ein Client kann Informationen zu einem Service beim Nameserver abfragen, welcher diesem dann mitteilt, wo der Service abrufbar ist. Der Client kann dann den Service direkt beim entsprechenden Server nutzen. Auf Wunsch kann der Cli-

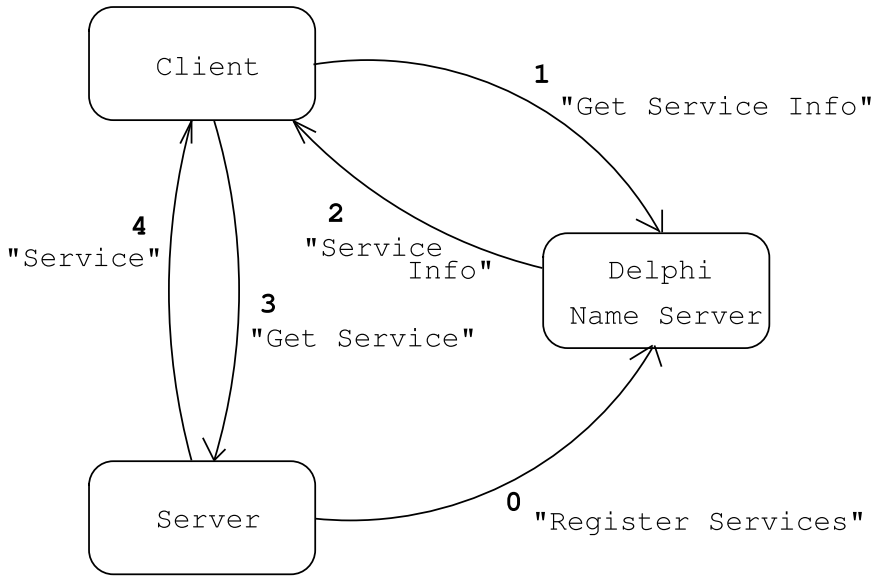

Abbildung 1.4: Ablauf von der Veröffentlichung von Informationen in DIM[9]

ent Änderungen an den veröffentlichten Daten mitgeteilt bekommen, sobald diese eintreten. DIM ist robust gegen Ausfälle von Servern, indem Clients erkennen können, dass ein Server ausfällt und wenn dieser wieder verfügbar ist.[9]

1.3 Anforderungen von CBM an das ECS

Die Aufgabe des ECS ist es, eine Vielzahl von ansonsten unabhängigen Systemen in einer globalen Sicht miteinander zu verbinden. Der Fokus des ECS liegt auf Zuverlässigkeit gegenüber maximaler Performanz, da die Aktionen wie das Konfigurieren von Systemen oder das Starten der Datennahme auch ohne den Overhead des ECS einige Sekunden bis zu mehreren Minuten dauern können.
Außerdem soll das ECS auch zusätzlich zur Benutzerschnittstelle Status und Statusänderungen des Systems auch dritten Programmen mitteilen können.
Ein großer Unterschied zwischen ALICE und dem CBM-Experiment besteht darin, dass die Detektoren in CBM selbst-getriggert sind und daher keine

Verwaltung von globalen Triggern im ECS nötig ist. Eine weitere Folge der selbstgetriggerten Auslese ist, dass die einzelnen Subsysteme stärker voneinander unabhängig sind als im ALICE-Experiment. Eine Eigenschaft, die aus diesem Vorteil gewonnen werden soll, ist, dass Detektoren ausfallen und neu konfiguriert werden können, ohne den Zustand der anderen Subsysteme zwangsläufig zu beeinflussen.
Wie auch im ALICE-ECS bietet es sich an, Systemzustände in *Finite State Machines* abzubilden, um bestimmte Operationen nur in bestimmten Systemzuständen zuzulassen.
Das ECS soll über ein *User Interface* steuerbar sein. Es bietet sich hierbei eine webbasierte Oberfläche an, wodurch mehrere Vorteile erreicht werden. Zum einen wird die Oberfläche hierdurch plattformunabhängig zum anderen wird auf der Client-Seite keine Installation weiterer Software benötigt. Außerdem ist auf Seite der Clients keine Versionsaktualisierung notwendig, wodurch Wartungsaufwand reduziert wird. Wie auch im ALICE-ECS soll es immer nur einem Benutzer möglich sein das Experiment zu steuern.
Das ECS soll Daten veröffentlichen können, sodass Komponenten des ECS oder dritte Anwendungen diese Daten abfragen können und Änderungen an diesen mitbekommen. Veröffentlichte Daten können zum Beispiel Zustände der *State Machines* sein.
Das Ziel dieser Arbeit ist es, zunächst einen Prototypen zu erstellen, welcher dann gegebenenfalls zu einem späteren Zeitpunkt weiterentwickelt werden kann. Daher sollen Komponenten generisch und leicht austauschbar entwickelt werden, um so eine zukünftige Weiterentwicklung zu erleichtern.

2 Konzeption

In diesem Kapitel geht es um die Konzeption des *Experiment Control Systems*, wie das System aufgebaut ist, aus welchen Komponenten es besteht und wie diese miteinander in Verbindung stehen ohne dabei auf genaue Implementierungsdetails einzugehen.

2.1 Aufbau des Systems

Abbildung 2.1 zeigt ein Beispiel für den Systemaufbau mit zwei Partitionen und fünf Detektoren. Die Verbindungen zwischen den Komponenten zeigen, welche Systeme miteinander kommunizieren.
Die Struktur des Systems orientiert sich am ALICE-ECS und ist hierarchisch in drei Ebenen unterteilt, die ECS-Ebene, die Partitionsebene und die Subsystemebene. Dabei werden auf ECS-Ebene eine Menge von Partitionen über den *Experiment Control Agent* (ECA) verwaltet werden, auf Partitionsebene jeweils eine Menge von Subsystemen über *Partition Control Agents* (PCA) und auf Subsystemebene jeweils ein Subsystem über einen *Controller Agent*.
Bei Subsystemen wird zwischen zwei Arten unterschieden. Zum einen gibt es Detektoren, welche immer zu genau einer Partition gehören, und zum anderen partitionsübergreifende globale Systeme wie zum Beispiel den FLES, welche Teil aller Partitionen sind. Der Grund, dass Detektoren sich nur in höchstens einer Partition befinden können ist, dass diese ansonsten von mehreren Partitionen gleichzeitig gesteuert würden, was diese wahrscheinlich

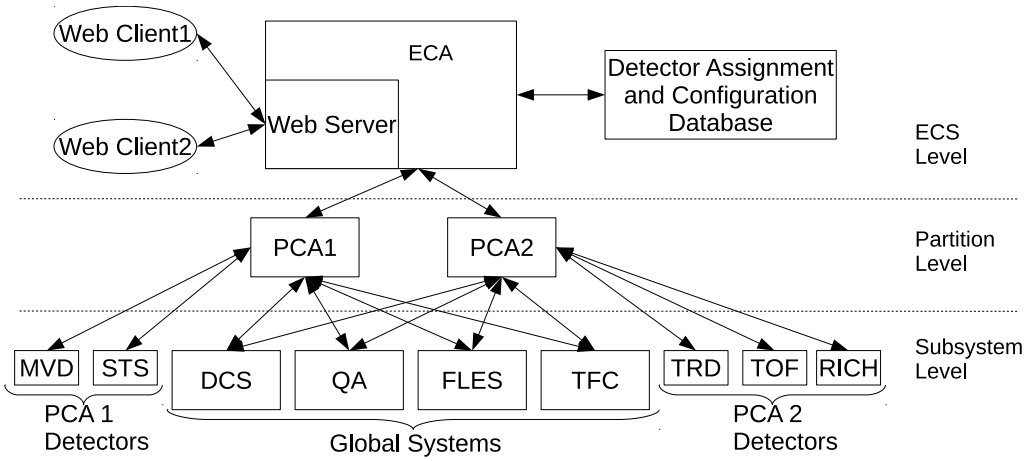

Abbildung 2.1: Ein Beispiel für Systemaufbau mit zwei Partitionen und fünf Detektoren.

nicht unterstützen werden. Globale Systeme wie der FLES müssen in allen Partitionen vorkommen, da ansonsten keine Datennahme möglich ist. Die Logik der Partitionierung muss hier systemintern abgebildet werden, darunter auch die Möglichkeit für verschiedene Systeme in unterschiedlichen Zuständen zu sein.
Subsysteme sind im Fall von Abbildung 2.1 die Detektoren MVD, STS, TRD, TOF und RICH, welche alle zu genau einer Partition gehören. DCS, QA, FLES und TFC sind partitionsübergreifende Systeme, welche in Verbindung mit allen Partitionen stehen.
Die Detektoren lassen sich in mehrere Partitionsmengen teilen, welche weitgehend unabhängig voneinander laufen. Dies soll ermöglichen, dass mehrere Experimente gleichzeitig und unabhängig voneinander ablaufen können oder dass fehlerhafte Detektoren in einer anderen Partition getestet werden können und nach der Fehlerbehebung wieder in das eigentliche Experiment eingebunden werden können.
ECA, PCAs und Subsysteme können auf verschiedenen Knoten nach dem Start eigenständig laufen. Hierdurch entsteht der Vorteil, dass man einzelne Komponenten im Fehlerfall neu starten kann, ohne dabei das komplette System neu starten zu müssen.

Systemzustände werden mit Hilfe von *Finite State Machines* (FSM) erfasst und gesteuert. Dies hat zum einen den Vorteil, dass die Systemabläufe der Hardware sauber in der Software abgebildet werden können, und zum anderen, dass in bestimmten Zuständen ungültige Operationen von vornherein ausgeschlossen werden können, um Fehlbedienungen zu vermeiden.
Finite State Machines existieren sowohl auf Partitionsebene als auch auf Subsystemebene. Die FSMs auf Subsystemebene arbeiten unabhängig voneinander, während die FSMs auf Partitionsebene in Abhängigkeit zu denen der Subsysteme stehen, welche der Partition zugeordnet sind. Da sich die FSMs der Subsysteme untereinander unterscheiden und viele systemspezifische Zustände haben können, wird die Zustandsmenge der Subsystem-FSMs auf Partitionsebene auf eine reduzierte Zustandsmenge abgebildet.
Wie man in Abbildung 2.1 sehen kann, ist ein Webserver Teil des ECA. Über diesen kann man über einen Webbrowser Systemzustände einsehen. Außerdem können über diese Oberfläche auch unter anderem Befehle über den ECA an die Partitionen verschickt werden oder Partitionen bearbeitet oder angelegt werden. Ein Webinterface hat den Vorteil, dass der Zugriff auf das System weitgehend plattformunabhängig ist, als auch, dass auf Seite des Clients – abgesehen vom Webbrowser – keine Installation weiterer Software und Versionsaktualisierung notwendig ist.

2.2 Systemkomponenten

In diesem Kapitel werden die bereits erwähnten Systemkomponenten genauer beschrieben.

2.2.1 *Experiment Control Agent*

Der *Experiment Control Agent* ist innerhalb des Systems einzigartig. Seine Hauptaufgabe besteht darin, die Partitionen und ihre Detektorzuweisungen zu verwalten. Um Daten über Detektoren und Partition persistent zu speichern, wird eine Datenbank verwendet. Dort wird neben der Zuweisung

der Detektoren zu Partitionen auch abgelegt, auf welcher Adresse und unter welchem Port die Partitionen und Subsysteme erreichbar sind. Diese Informationen werden vom ECA an die Subsysteme bereitgestellt, welche diese zum Starten benötigen.
Der ECA läuft im selben Prozess wie der Webserver, über welchen das *Web User Interface* läuft, und leitet von diesem erhaltene Kommandos an den entsprechenden PCA weiter. Der Betrieb beider Komponenten im selben Prozess verhindert eine weitere Kommunikationsebene, welche keinen weiteren Sinn hat, als Daten zwischen Web-Client und ECA über den Webserver weiterzuleiten und für zusätzlichen Kommunikations-Overhead zu sorgen.
Von den PCAs erhält der ECA alle PCA- und Subsystem-Zustandsänderungen, speichert diese und leitet sie an die Web-Clients weiter. Eine direkte Kommunikation zwischen dem ECA und Detektoren ist nur in Sonderfällen vorgesehen. Der ECA kommuniziert in erster Linie nur mit den PCAs.
Es existiert die Möglichkeit Detektoren keiner Partition zuzuweisen, zum Beispiel weil diese inaktiv oder defekt sein könnten. Da auch diese trotzdem im System erfasst sein sollten, existiert ein leichtgewichtiger PCA namens *Unmapped Detector Controller*, welcher Teil des ECA ist. Dieser Controller empfängt Updates der unzugewiesenen Detektoren, kann aber keine Kommandos außer Abbruch-Kommandos an diese versenden und besitzt keine eigene *State Machine*.
Der ECA versendet in regelmäßigen Zeitabständen Ping-Signale an alle PCA und bemerkt so eventuelle Verbindungsabbrüche und sendet diese Information an das Webinterface weiter. Ebenfalls hat der ECA die Möglichkeit, PCA oder *Controller Agents* auf ihren entsprechenden Knoten zu starten oder zu beenden.

2.2.2 Webserver

Die Aufgabe des Webservers ist das Bereitstellen der Webnutzer-Oberfläche. Hierüber kann ein Benutzer den Zustand des Systems betrachten, Befehle versenden, Detektoren und Partitionen anlegen oder löschen und Detektoren

zwischen Partitionen verschieben. Auch Konfigurationen für Subsysteme können hierüber angelegt werden. Über den Webserver läuft auch das Verwalten der Benutzerberechtigungen. Nur jeweils ein Benutzer kann neue Detektoren und Partitionen anlegen. Genauso kann auch nur jeweils ein Benutzer eine kontrollierende Rolle über einen PCA einnehmen. Der Webserver läuft im selben Prozess wie das ECS und kann so direkt auf alle benötigten Daten des aktuellen Systemzustands zugreifen.

2.2.3 *Partition Control Agent*

Der *Partition Control Agent* (PCA) verwaltet eine Partition von Detektoren und hält den Zustand seiner Partition in einer *State Machine* fest, welche abhängig von den Zuständen der *Subsystem State Machines* ist.
Dabei kommuniziert er mit den *Controller Agents* seiner zugewiesenen Detektoren und den *Controller Agents* von partitionsübergreifenden Subsystemen. Die Menge der übergreifenden Systeme ist fest, während die Anzahl Detektoren beliebig ist und sich auch zur Laufzeit ändern kann.
Der PCA bekommt von allen seinen *Controller Agents* auftretende Zustandsänderungen mitgeteilt, aus welchen er dann einen globalen Zustand für seine Partition bestimmen muss. Alle globalen und *Controller Agent* Zustandsänderungen werden dem ECA und gegebenenfalls weiteren externen Systemen mitgeteilt.
Der PCA kann auf einem anderen Rechner betrieben werden als der ECA oder die *Controller Agents*. Zugewiesenen *Controller Agents* werden in regelmäßigen Zeitabständen Signale zugesendet um so Ausfälle zu bemerken.

2.2.4 *Controller Agent*

Der *Controller Agent* verwaltet ein Subsystem wie zum Beispiel einen Detektor oder den FLES. Er bildet das direkte Interface vom ECS zum Subsystem und implementiert dessen ECS-Logik. Der aktuelle Zustand des Systems wird in

einer *State Machine* festgehalten, welche unabhängig vom globalen Zustand der Partition ist. Im Falle eines partitionsübergreifenden Systems verwaltet der *Controller Agent* eine *State Machine* pro Partition, da diese Teil mehrerer Partitionen sind. Hierbei können die *State Machines* unabhängig voneinander in verschiedenen Zuständen sein. Beispiele, in denen unterschiedliche Zustände auftreten, könnten sein, dass für eine Partition keine Hardwareressourcen mehr vorhanden sind oder dass Partitionen unterschiedliche Konfigurationsparameter übergeben bekommen haben.
Der *Controller Agent* empfängt Kommandos von seinem PCA und führt dann eine entsprechende Aktion aus. Die Implementierung dieser Aktionen kann sich von Subsystem zu Subsystem unterscheiden. Der *Controller Agent* bemerkt Zustandsänderungen seines Subsystems und teilt diese mit seinem PCA.

3 Implementierung

In diesem Kapitel geht es um die Implementierung des *Experiment Control Systems* (ECS). Hierzu werden zunächst Implementierungsdetails aller Komponenten des ECS gezeigt. Danach wird erläutert, wie die Kommunikation zwischen den Subsystemen abläuft. Zuletzt wird ein Überblick über die Nutzeroberfläche gegeben.
Das ECS ist in der Programmiersprache Python implementiert. Diese Sprache wurde aus mehreren Gründen gewählt. Ein Grund ist, dass sie eine leichte Syntax hat und so gut lesbaren und verständlichen Programmcode bietet, was einen Vorteil bietet, wenn der Prototyp in Zukunft weiterentwickelt oder geändert werden soll. Außerdem ist Python zusammen mit C++ eine weit verbreitete Sprache im wissenschaftlichen Umfeld. Ein Nachteil von Python im Vergleich zu C++ ist, dass Python in der Ausführungszeit etwas langsamer läuft. Da der Fokus des ECS allerdings nicht auf Performanz liegt, wird dieser Nachteil zu Gunsten der genannten Vorteile in Kauf genommen.

3.1 *Experiment Control Agent*

Abbildung 3.1 zeigt den Aufbau des ECA mit seinen einzelnen Komponenten und wie diese miteinander in Verbindung stehen. Eine Komponente ist der Webserver, welcher gemeinsam mit dem ECA in einem Prozess läuft. Dieser dient als Schnittstelle zum *Web User Interface* und zur Verwaltung von Benutzerrechten. Vom Web-Client gesendete Befehle werden vom Webserver empfangen, welcher dann direkt über die PCA-Handler den Befehl zum PCA

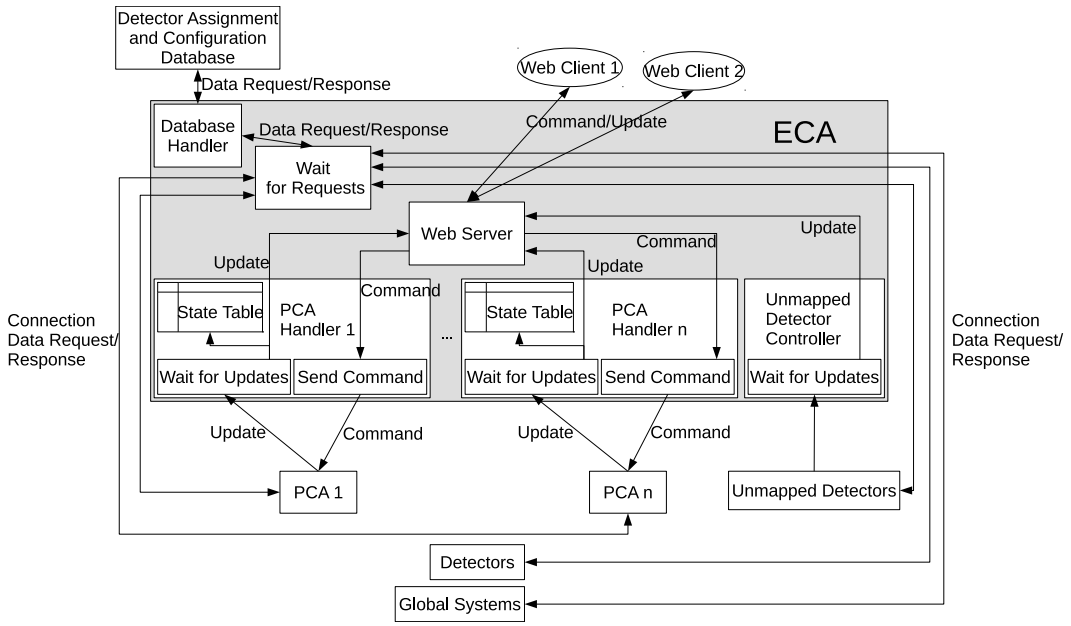

Abbildung 3.1: Die Struktur des *Experiment Control Agent* (ECA) mit seinen internen Komponenten und wie diese mit externen ECS-Komponenten in Verbindung stehen.

weiterleitet. Im Abschnitt 3.5 wird genauer auf den Webserver eingegangen. Im Folgenden werden weitere Komponenten erläutert. Zum Schluss dieses Kapitels wird die Verwaltung von Partitionen und Detektoren durch den ECA dargestellt.

3.1.1 Datenbank

Eine Aufgabe des ECA ist es, Partitionen und Detektorzuweisungen zu verwalten. Um diese persistent zu speichern, bietet es sich an eine Datenbank zu verwenden. Für den ECS-Prototyp wird hierbei eine SQlite-Datenbank verwendet. Die Kommunikation mit der Datenbank läuft über eine Schnittstelle ab, welche in Abbildung 3.1 unter dem Namen Database-Handler abgebildet ist. Die Ergebnisse einer Datenbankabfrage werden in festgelegten Objekten abgelegt, um innerhalb des Programmcodes leichter auf einzelne Datenelemente zugreifen zu können und die Ergebnisse der Abfrage leichter zwischen ECS-Komponenten verschicken zu können. Das Kapseln der Datenbanklogik

in einer eigenen Klasse erreicht zwei Vorteile. Zum einen wird die Abfragelogik der Datenbank streng vom Rest des Programmcodes getrennt, zum anderen kann man die aktuell verwendete SQlite-Datenbank relativ leicht mit einer anderen Datenbank austauschen.

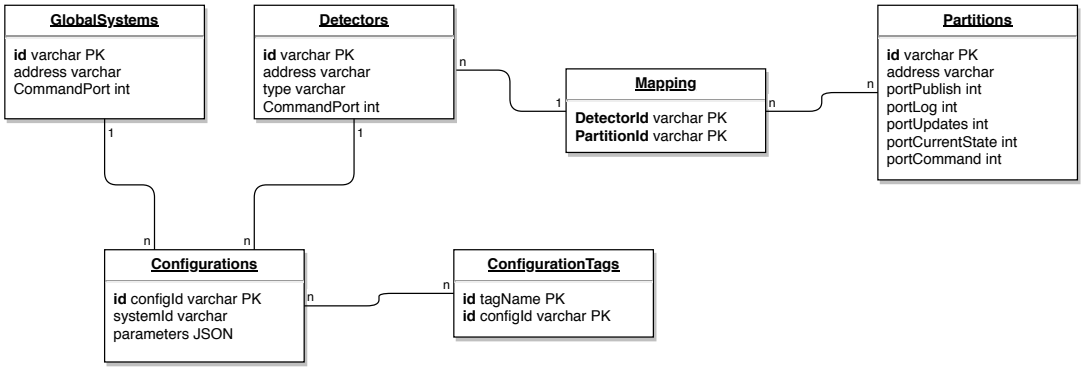

Abbildung 3.2: Datenbankschema für das ECS

In Abbildung 3.2 sieht man das Schema für die Datenbank. In den Tabellen „Detector“, „Partition“ und „GlobalSystems“ werden die PCAs beziehungsweise Subsysteme mit einer Id, Adresse und Ports abgelegt. In der Tabelle „Mapping“ werden die Zuordnungen von Detektoren zu Partitionen festgelegt. Ein Detektor kann nur jeweils einer Partition zugeordnet werden, weshalb „DetectorId“ höchstens einmal in Mapping vorhanden sein kann. Für Detektoren existiert ein Attribut „type“, mit welchem der Typ des Detektors festgelegt wird. Dieses Attribut wird genutzt, um dem Detektor eine entsprechende Objektklasse zuordnen zu können.
Bei der Konfiguration von Subsystemen muss das ECS diesen mitteilen, wie und wofür diese sich konfigurieren sollen. Dies geschieht über sogenannte Konfigurations-Tags. Durch das Ablegen dieser Tags in der ECS-Datenbank entsteht der Vorteil, dass diese in einer zentralen Instanz verwaltet werden und nicht auf mehrere Datenverwaltungssysteme verteilt sind.
In der Tabelle „Configurations“ können für Subsysteme Konfigurationen mit Parametern unter einer Id abgelegt werden. Da sich diese Parameter sowohl in Art als auch Menge unterscheiden können, kann man für diese in einer relationalen Datenbank keine feste Tabellenstruktur festlegen. Daher werden die Parameter als JSON-Objekt abgelegt. Da eine globale Konfiguration für eine Partition aus mehreren Konfigurationen für Subsysteme besteht, können

mehrere Konfigurationen zu einem globalen Konfigurations-Tag zusammengefasst werden, welches in der Tabelle „ConfigurationTags“ abgelegt wird. Konsistenz zwischen den Tabellen beim Erstellen, Löschen und Ändern von Datensätzen wird durch Constraints und Trigger gewährleistet.
Andere Komponenten des ECS oder gegebenenfalls auch dritte Anwendungen haben die Möglichkeit, Anfragen an den ECA zu senden, um Detektorzuordnungen oder Adress- und Port-Daten zu erfahren. Der ECA hat hierzu einen Thread („Wait for Requests“ in Abbildung 3.1), welcher auf einem Socket lauscht, empfangene Anfragen an die Datenbank weiterleitet und dann dessen Antwort an den Anfragesteller zurück sendet. Innerhalb des ECS sind diese Anfragen zum Beispiel für den PCA notwendig, um seine Detektorzuweisung zu erfahren.

3.1.2 PCA-Handler

Da der ECA mehrere PCAs verwaltet, ist es sinnvoll, die Verwaltung eines PCAs in einer eigenen Klasse zu kapseln. Dies geschieht über die PCA-Handler, diese dienen als Schnittstelle zwischen PCA und ECA. Pro Partition wird ein PCA-Handler erzeugt.
Eine wichtige Aufgabe des PCA-Handlers ist das Sammeln von Status-Updates und Log-Nachrichten seines PCAs und das Weiterleiten von diesen an die verbundenen Web-Clients. Jeder PCA-Handler verbindet sich als Subscriber mit dem jeweiligen Publisher-Socket des PCA und empfängt über diesen alle Status-Updates der Partition. Erhaltene Zustandsänderungen werden dann über einen Websocket an die Web-Clients weitergeleitet. Log-Nachrichten werden sowohl in eine Datei geschrieben als auch vom ECA in einer Queue zwischengespeichert. Die zwischengespeicherten Log-Nachrichten können von einem Web-Client abgefragt werden, um diese auch nach dem Neuladen einer Webseite weiterhin anzeigen zu können. Die Anzahl der gespeicherten Nachrichten ist frei konfigurierbar. Sobald die Queue ihre maximale Länge erreicht hat, wird die älteste Nachricht verworfen. Durch das Verwerfen älterer Nachrichten soll erreicht werden, dass die Menge der zu übertragenden Log-Daten bei einer Anfrage nicht beliebig groß werden kann.

Über den PCA-Handler werden auch Kommandos wie zum Beispiel eine Aufforderung zur Konfigurierung an den PCA weitergeleitet. Um Verbindungsabbrüche zu bemerken, versendet der PCA-Handler periodisch Pings an seinen PCA. Im Falle eines unbeantworteten Pings werden Web-Clients hierüber informiert.

3.1.3 *Unmapped Detector Controller*

Da es möglich ist, dass Detektoren existieren, welche keiner Partition zugewiesen sind, müssen auch diese vom System erfasst werden. Hierfür existiert ein leichtgewichtiger PCA namens *Unmapped Detector Controller* (UDC), welcher Teil des ECA ist.
Die Aufgabe des UDC ist es, unzugewiesene Detektoren zu erfassen, ohne eine Kontrolllogik zu besitzen. Er besitzt keine eigene *State Machine*, sondern empfängt nur Updates von Detektoren und überprüft mit Pings, ob diese erreichbar sind. Zustandsänderungen oder Verbindungsänderungen werden veröffentlicht. Außer Abbruch-Befehlen können keine Befehle an die unzugewiesenen Detektoren versendet werden.

3.1.4 Verwalten von Partitionen und Detektoren

Der ECA hat auch die Möglichkeit, neue Partitionen und Detektoren anzulegen. Im Falle einer Partition wird hierfür eine Id, Adresse und Ports für die Sockets benötigt. Im Falle eines Detektors werden ebenfalls Id, Adresse und ein Port benötigt. Hinzu kommt der Typ des Detektors, welcher einer Implementierung des *Detector Controllers* (siehe 3.3) und einem *Partition Component Object* (siehe 3.2.1) zugewiesen sein muss. Optional kann eine Partition für den Detektor ausgewählt werden, welcher dieser dann – falls möglich – hinzugefügt wird. Falls keine Partition gewählt wird, wird der Detektor automatisch dem UDC des ECA zugewiesen.
Der ECA hat die Möglichkeit, Detektorzuweisungen zur Laufzeit zu ändern.

Dies ist zum Beispiel notwendig, wenn man einen zuvor getesteten Detektor wieder in das Hauptsystem einbinden möchte. Es gibt drei Möglichkeiten einer Zuweisungsänderung. Ein Detektor kann:

- zwischen zwei Partitionen verschoben werden
- von einer Partition zum UDC verschoben werden
- vom UDC in eine Partition verschoben werden

Das Verschieben eines Detektors während einer Datennahme oder einer Konfiguration könnte zu Fehlern innerhalb der beteiligten Subsysteme führen. Zur Vorbeugung von Fehlern ist es daher grundsätzlich nur möglich, einen Detektor zu verschieben, falls die beteiligten Partitionen zu diesem Zeitpunkt kein System konfigurieren und sich nicht in einer Datennahme befinden.
In den Partitionen wird vor dem Verschieben des Detektors jeweils ein Lock-Flag gesetzt, welches verhindert, dass während des Vorgangs Befehle entgegengenommen werden. Nach Ende des Verschiebevorgangs werden diese Flags wieder zurückgesetzt. Auf diese Weise wird verhindert, dass auf Partitionen während des Verschiebens Konfigurationen oder Datennahmen gestartet werden.
Das Verschieben eines Detektors ist ein sehr kritischer Vorgang, da im Fehlerfall unter anderem Detektoren zwei Partitionen zugewiesen sein könnten oder komplett aus dem System verschwinden könnten. Daher läuft das Verschieben als Transaktion ab, welche entweder komplett oder gar nicht ausgeführt wird. Falls während des Verschiebens ein Fehler auftritt, wird ein Rollback ausgeführt, welches das System in den Ausgangszustand zurückversetzt.
Für den Fall, dass das Verschieben zwischen zwei PCAs ausgeführt wird, werden folgende Schritte ausgeführt:

1. Lock-Flags von Partitionen setzen
2. Zuweisung in Datenbank ändern
3. Detektor von altem PCA entfernen
4. Detektor neuem PCA hinzufügen

5. globale Systeme informieren

6. *Detector Controller* über Änderung informieren

7. Lock-Flags von Partitionen zurücksetzen

Falls der Detektor vorher keiner Partition zugewiesen war oder keiner neuen Partition zugewiesen wird, wird in Schritt 3 beziehungsweise 4 der UDC involviert. Der ECA merkt sich, welche Schritte bereits erfolgreich ausgeführt wurden und macht diese im Fehlerfall rückgängig, um das System in seinen Ausgangszustand zurückzuversetzen.
Da es sich bei PCAs und *Controller Agents* um eigenständige Prozesse in einem Netzwerk handelt, müssen beim Start des ECS gegebenenfalls eine Vielzahl von Prozessen gestartet werden. Außerdem kann es im Fehlerfall eines *Controller Agents* oder PCAs notwendig sein, diesen neu zu starten. Im ECA wurde daher die Möglichkeit implementiert Clients wie den PCA, *Detector Controller* oder *Controller Agents* für globale Systeme auf ihren in der Datenbank zugewiesenen Adressen starten. Dies geschieht über eine Library namens Paramiko, welche es ermöglicht in Python über SSH Befehle auf anderen Rechnern auszuführen. Voraussetzung hierfür ist, dass der Benutzer, welcher den ECA startet, die anzusprechenden Knoten in seiner Known-Hosts-Liste hat und sich auf diesen ohne Passworteingabe mithilfe eines SSH-Keys einloggen kann. Ob PCAs und *Controller Agents* automatisch gestartet werden sollen, lässt sich in einer Konfigurationsdatei festlegen.

3.2 *Partition Control Agent*

Der *Partition Control Agent* (PCA) steuert und überwacht eine Partition von Detektoren mithilfe einer globalen *State Machine*. Abbildung 3.3 zeigt den Aufbau eines PCA.

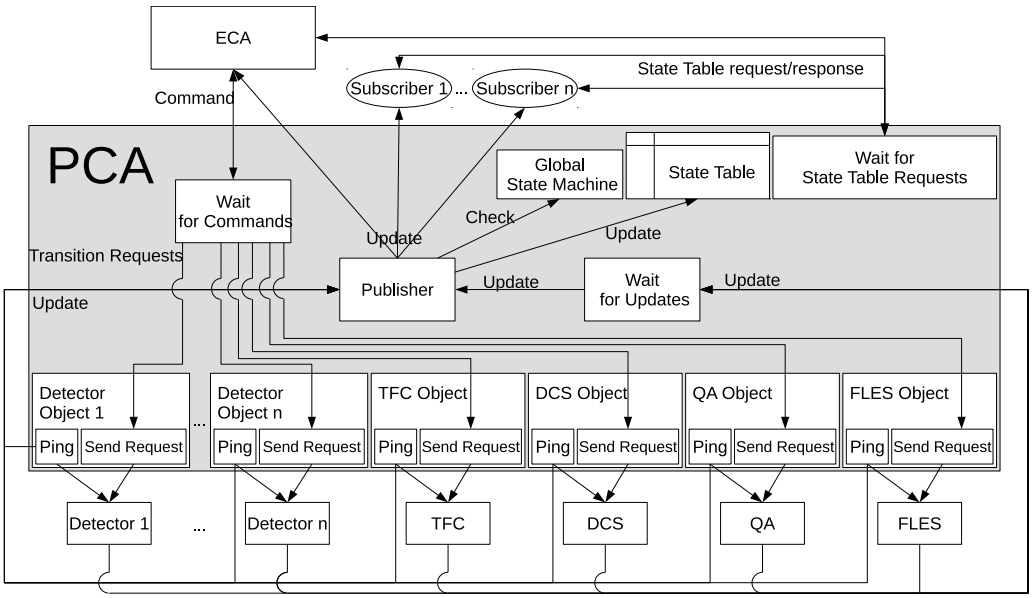

Abbildung 3.3: Der Aufbau eines *Partition Control Agents* mit seinen internen Komponenten und wie diese mit externen ECS-Komponenten in Verbindung stehen.

3.2.1 *Partition Component Object*

Der PCA verwaltet mehrere Subsysteme. Wie auch beim ECA ist es sinnvoll, diese Subsysteme jeweils in einem eigenen Objekt zu kapseln. Zu diesem Zweck existiert die Klasse namens *Partition Component Object* (PCO). Für jedes Subsystem wird jeweils ein PCO erstellt, welches dann als Schnittstelle zu dessen *Controller Agent* fungiert.
Abbildung 3.4 zeigt ein Klassendiagramm zu dem PCO. Bei Subsystemen wird zwischen Detektoren und globalen Systemen wie TFC, DCS, FLES und QA unterschieden. Detektoren erben von einer Klasse für Detektoren, welche allgemeine Detektor-Funktionalität enthält. Analog erben globale Systeme von einer Klasse für globale Systeme. Sowohl PCO für Detektoren als auch für globale Systeme erben von einer Partitions-Objektklasse, welche die allgemeinste Funktionalität für beide enthält.
In den Klassen für die expliziten Subsysteme muss die Übergangsfunktionalität implementiert werden, zum Beispiel wie diese sich bereit zur Datennahme machen sollen und wie man sie wieder zurücksetzen kann. Der Großteil dieser Funktionalität liegt zwar in den *Controller Agents*, aber so können zum Bei-

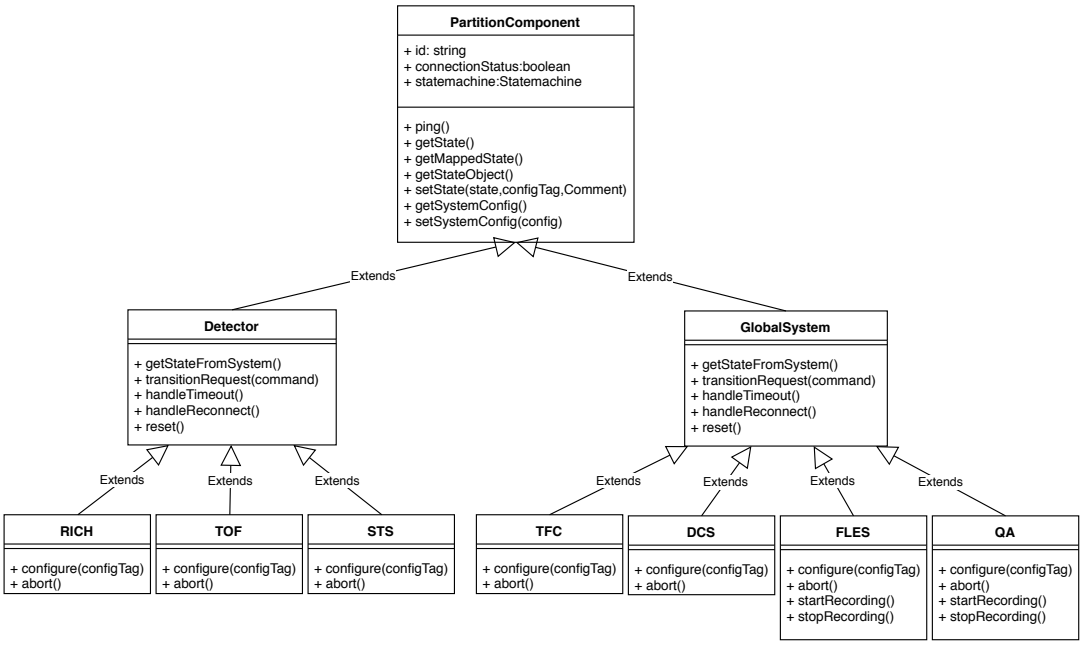

Abbildung 3.4: Klassendiagramm für Partition-Komponenten

spiel im aktuellen Zustand unmögliche Transitionen bereits auf PCA-Ebene abgefangen werden.
Das genaue Verhalten einzelner Subsysteme kann sich voneinander unterscheiden. Daher kann sich auch die genaue Implementierung der PCO von Subsystem zu Subsystem unterscheiden. Diese Vererbungshierarchie wurde so gewählt, damit die Schnittstellen-Klassen möglichst generisch bleiben, um leicht neue Subsysteme hinzufügen zu können. PCOs besitzen zwar wie auch *Controller Agents* eine *State Machine*, benutzen diese allerdings nur, um den Zustand festzuhalten. So kann vor Transitions-Anfragen bestimmt werden, ob eine vom Benutzer geforderte Transition gültig ist. Bei einer empfangenen Zustandsänderung wird der Zustand der *State Machine* einfach nur gesetzt und nicht durch Zustandsübergänge überführt. Da die Übergänge bereits auf Detektorebene durchgeführt werden, ist es nicht notwendig, diese auf PCA-Ebene zu wiederholen. Außerdem könnten dem PCA im Falle von Verbindungsabbrüchen ein oder mehrere Übergänge verloren gehen.
Die Zuordnung von Detektoren zu ihrem jeweiligem PCO geschieht über das Typen-Attribut, welches in der Datenbank für den Detektor festgelegt ist.

3.2.2 Verwaltung von Zuständen

Der PCA hat zum einen die Aufgabe, sich seinen globalen Zustand aus den Zuständen seiner Subsysteme herzuleiten und zum anderen die Aufgabe, aktuelle Systemzustände anderen Systemen bereitzustellen. Der PCA muss, um dies zu erreichen, Zustände und Zustandsänderungen seiner Subsysteme erfassen und sich diese abspeichern.
Der PCA speichert die Zustände der *State Machines* aller Subsysteme sowie seinen eigenen Zustand in einer Statustabelle. Bei dieser Tabelle handelt es sich um einen *Key Value Store*, wobei der Key jeweils der Id der Partition oder des Subsystems entspricht. Die Values enthalten jeweils folgende Daten:

- Sequenznummer
- Zustand der *State Machine* aus PCA-Sicht
- Zustand der *State Machine* aus Subsystem-Sicht
- Id der aktuellen Konfiguration des Systems oder der Partition
- Anmerkung zum Zustand für das Webinterface

Auf die Sequenznummer wird in Abschnitt 3.4 genauer eingegangen. Der Unterschied zwischen einem Zustand aus PCA-Sicht und aus Subsystem-Sicht wird später in diesem Kapitel in 3.2.3 betrachtet.
Die Zustände der Subsysteme werden beim Start der PCAs oder nach einem Verbindungsabbruch bei den jeweiligen *Controller Agents* abgefragt.
Status-Updates über Zustandsänderungen der Subsysteme werden auf einem eigens dafür vorgesehenen Socket empfangen. Empfangene Updates werden in die Queue des Publisher-Threads gelegt.
Der Publisher hat die Aufgaben, die Status-Tabelle und die *State Machines* des PCO zu aktualisieren, Status-Updates zu veröffentlicht und zu prüfen, ob eine globale Statusänderung vorliegt. Die Verarbeitung von Updates wird in einen extra Thread verlegt, da Status-Updates nicht nur von empfangenen Updates von Subsystemen erzeugt werden können, sondern auch durch Verbindungsabbrüche, Entfernen und Hinzufügen von Detektoren oder durch globale

Statusänderungen. Durch das Abarbeiten von Updates in einer Queue wird sichergestellt, dass alle Updates sequentiell betrachtet werden. Ein paralleles Verarbeiten von Updates könnte Fehleranfälligkeit durch Race-Conditions verursachen und das System durch benötigte Thread-Synchronisation verkomplizieren. Daher wurde an dieser Stelle Simplizität und Stabilität der Performanz bevorzugt.
Zudem existiert ein Thread, welcher mit einem Socket auf Anfragen an die gesamte Statustabelle des PCA wartet und diese dann mit einem aktuellen Snapshot seiner Statustabelle beantwortet. Diese Anfragen sind notwendig, um einem neuen Subscriber den aktuellen Stand mitzuteilen.

3.2.3 Zustands-Mapping

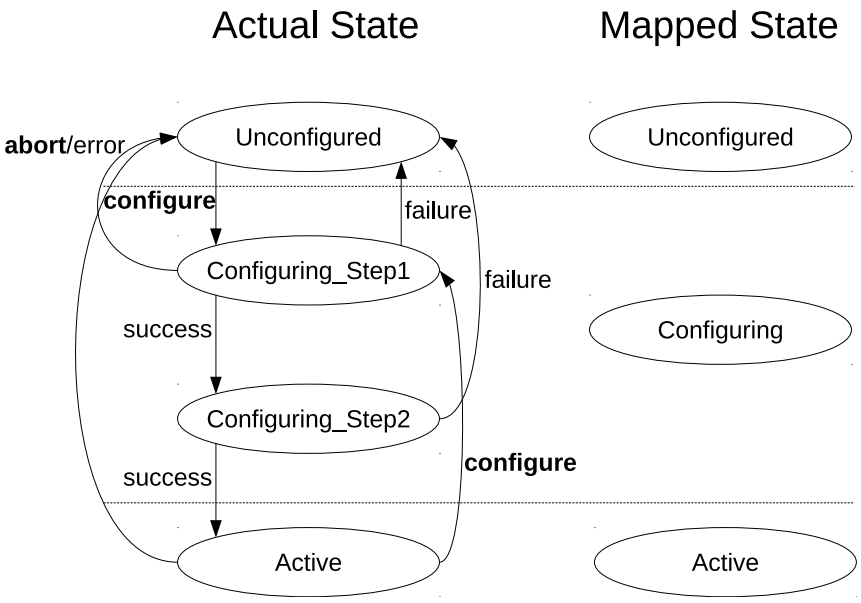

Abbildung 3.5: Beispiel für Zustands-Mapping

Der PCA bildet seinen aktuellen Systemzustand in einer für die Partition globalen *Finite State Machine* (FSM) ab. Der Zustand dieser *State Machine* hängt zu einem großen Teil von den Zuständen der Subsysteme ab, welche wiederum selbst eine *State Machine* besitzen. Da alle Subsysteme eine individuelle *State Machine* besitzen können, kann die zu betrachtende Zustandsmenge für den

PCA sehr groß werden, wenn dieser den globalen Zustand bestimmen möchte. Um dieses Problem zum lösen, wird die Zustandsmenge von Subsystemen auf eine kleinere abgebildet (Zustands-Mapping). In gewisser Weise wird so die *State Machine* des Subsystems so für den PCA vereinfacht.
In Abbildung 3.5 sieht man ein Beispiel für ein Zustands-Mapping. Links befindet sich eine *State Machine* und rechts das Zustands-Mapping, wobei die Zustände „Configure_Step1“ und „Configure_Step2“ auf den Zustand „Configure“ abgebildet werden. Das System in diesem Beispiel unterteilt seine Konfiguration in zwei Schritte. Für den PCA ist jedoch nur relevant, dass das System gerade dabei ist, sich zu konfigurieren.
Die Abbildung der Zustände wird in einer CSV-Datei festgelegt, welche beim Start des PCAs eingelesen wird.

3.2.4 *Finite State Machine* für PCAs

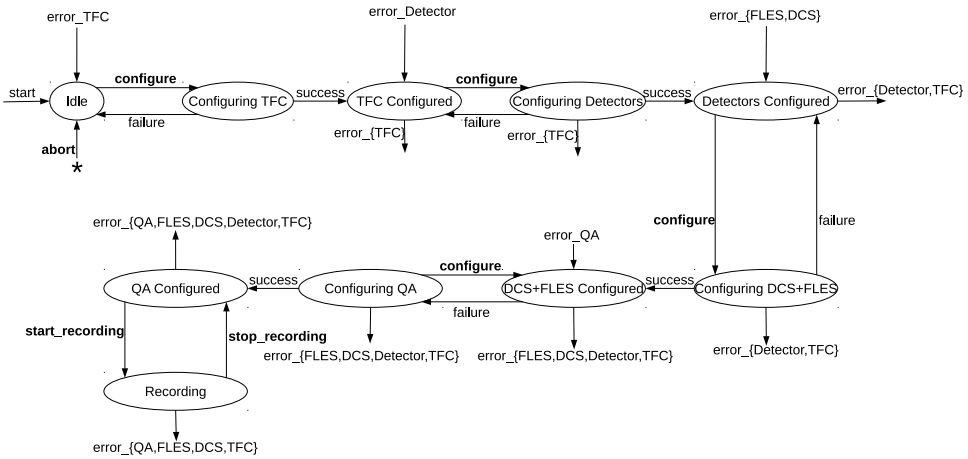

Abbildung 3.6: Die *State Machine* für einen PCA

In Abbildung 3.6 sieht man die verwendete globale *Finite State Machine* (FSM) für die PCAs. Der Startzustand ist der Zustand „Idle“. Die FSM ist hierarchisch aufgebaut, da manche Subsysteme zum Konfigurieren erfordern, dass andere Subsysteme bereits konfiguriert sind. Das TFC muss vor den Detektoren konfiguriert werden, da die Detektoren ihren genommenen Daten sonst keine Zeitstempel hinzufügen können. DCS und FLES benötigen, dass es Detektoren

gibt, welche Daten nehmen können und QA benötigt Daten, welche vom FLES geliefert werden. Zuletzt wird zwischen einem Zustand unterschieden, in welchem das System theoretisch bereit wäre, eine Datennahme zu starten und einem Zustand, in welchem Daten tatsächlich erfasst und auf Festplatten gesichert werden.
Für jedes Subsystem existieren zwei Zustände, einer, in welchem es konfiguriert wird und einer, in welchem es fertig konfiguriert ist. Das Konfigurieren der Subsysteme hat jeweils einen eigenen Zustand, um eine bessere Übersicht über den aktuellen Status zu geben, da das Konfigurieren gegebenenfalls lange Zeit dauern kann und man so sehen kann, dass sich das System gerade in einem Zustand befindet, in welchem es konfiguriert wird. Außerdem wird in dem konfigurierenden Zustand abgefangen, dass Benutzer Konfigurationsbefehle senden, obwohl das System sich noch in einer Konfiguration befindet. Bei Erfolg der Konfiguration geht die FSM in den entsprechenden Configured-Zustand des konfigurierten Subsystems über, beziehungsweise bei Misserfolg zurück in den Vorgänger-Zustand.
Falls bei einem Subsystem ein Fehler auftritt, wird das System in den Zustand vor der Konfiguration des Systems gesetzt. Wenn man sich zum Beispiel in „QA_Configured befindet“ und ein Fehler im TFC auftritt, wird die Transition „error_TFC“ ausgelöst und das System wechselt in den Zustand „Idle“, in welchem das TFC wieder konfiguriert werden muss. Von hier aus muss die Hierarchie der *State Machine* theoretisch erneut durchlaufen werden, es müssen also auch nachfolgende Systeme des ausgefallenen Systems neu konfiguriert werden. Je nach Implementierung können die Subsysteme allerdings selbst entscheiden, ob bei eingehendem „configure“-Kommando ein erneutes Konfigurieren notwendig ist und gegebenenfalls direkt einen „success“-Übergang melden. Hierdurch soll erreicht werde, dass ein Subsystem neu konfiguriert werden kann, ohne zwangsläufig den Konfigurationszustand anderer Subsysteme zu beeinflussen.
Zustandsübergänge können entweder durch den Benutzer (fettgedruckt Übergänge in 3.6) oder durch ein Ereignis ausgelöst werden. Ein Ereignis kann zum Beispiel sein, dass ein System meldet, dass es mit seiner Konfigurierung fertig ist. Übergänge wie „configure“ hingegen werden durch Kommandos des Benutzers ausgelöst.

Die globale *State Machine* wird, wie alle anderen *State Machines* auch, in einer CSV-Datei abgelegt. In einer Zeile dieser Datei steht jeweils ein Zustand, ein Übergang und ein Folgezustand. Beim Start eines PCA wird die entsprechend benötigte Datei eingelesen und aus dieser der Übergangsgraph der *State Machine* gebildet. Dadurch wird erreicht, dass die *State Machine* flexibel geändert werden kann, ohne dabei den Programmcode selbst zu ändern.

3.2.5 Transitions-Logik für die globale *State Machine*

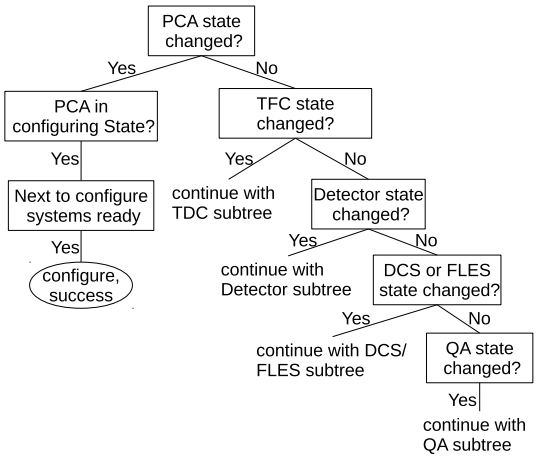

Abbildung 3.7: Entscheidungsbaum zur Ermittlung von globalen Übergängen der *State Machine*

Der PCA muss den globalen Zustand der Partition aus den Zuständen seiner Subsysteme bestimmen. Für jede Subsystem- oder globale Zustandsänderung wird im Publisher-Thread geprüft, ob eine Transition der globalen *State Machine* durchgeführt werden muss. Die Überprüfung wird anhand eines Entscheidungsbaumes durchgeführt. Das Verwenden eines Entscheidungsbaumes hat den Vorteil, dass so ein strukturierter und leicht nachvollziehbarer Programmcode entsteht, welcher bei Bedarf relativ leicht erweiter- oder änderbar ist. Um den Baum möglichst einfach zu halten, kommt an dieser Stelle das Zustands-Mapping zum Einsatz. Für Subsysteme müssen nur der nichtaktive, konfigurierende, aktive und im Fall von QA und FLES der zusätzlich aufzeichnende Zustand betrachtet werden.

Die Abbildung 3.7 zeigt die oberste Ebene des Entscheidungsbaums. Hier wird betrachtet, von welchem Subsystem die Zustandsänderung ausging. Danach wird die Entscheidungsfindung im entsprechenden Teilbaum für das Subsystem fortgesetzt. Die verschiedenen Teilbäume sind in den Abbildungen 3.8 bis 3.11 zu sehen. Die Blattknoten der Bäume zeigen Übergänge, welche in der PCA *State Machine* (Abbildung 3.6) ausgelöst werden. Nicht in den Abbildungen gezeigte Entscheidungspfade lösen keine Transition der globalen *State Machine* aus.
Globale Zustandsänderungen lösen, wie auch Änderungen von Subsystemen, eine Prüfung des aktuellen Zustands aus (linker Pfad in 3.7). Hierbei wird in nicht-konfigurierenden Zuständen geprüft, ob die nächsten Systeme in der Hierarchie der *State Machine* bereits korrekt konfiguriert sind. Falls dies der Fall ist, geht der PCA automatisch in der Hierarchie eine Ebene nach oben. Hierdurch ist es dem PCA möglich, nach einem Neustart den korrekten Zustand in der Hierarchie der *State Machine* wiederherzustellen.
Jeder der Teilbäume in 3.8 bis 3.11 repräsentiert jeweils eine Ebene der Hierarchie der globalen *State Machine* und wird jeweils für ein oder eine Menge von Subsystemen benutzt.
Die Teil-Entscheidungsbäume sind vom Prinzip ähnlich aufgebaut. Zunächst wird geprüft, ob der PCA gerade dabei ist, die entsprechende Hierarchieebene zu konfigurieren. Im Ja-Fall wird geprüft, ob alle Subsysteme mit der Konfiguration fertig sind. Sobald sich kein konfigurierendes Subsystem in der Ebene befindet, wird geprüft, ob alle Systeme der Ebene bereit sind. Falls dies der Fall sein sollte, wird eine Erfolg-Transition ausgeführt, beziehungsweise ansonsten eine Misserfolg-Transition.
Wenn der PCA sich nicht im konfigurierenden Zustand befindet, wechselt dieser dorthin, sobald ein Subsystem der Hierarchieebene sich zu konfigurieren beginnt.
Es kann passieren, dass Subsysteme bereit werden, ohne sich vorher zu konfigurieren, zum Beispiel wenn ein Verbindungsabbruch eintritt und sich ein Subsystem im Aktiv-Zustand wieder verbindet. Der PCA muss auch in diesem Fall seinen korrekten Zustand finden können und – falls notwendig – die nächste Konfiguration überspringen. Wenn der genannte Fall auftritt und sich der PCA nicht im konfigurierenden Zustand befindet, wechselt der PCA in die

nächste Hierarchieebene, falls alle anderen Subsysteme der Ebene bereit sind und sich der PCA in dem Zustand befindet, in welchem die Subsysteme als Nächstes konfiguriert werden müssten. Falls sich ein PCA beispielsweise im Zustand „TFC Configured“ befindet und ein Detektor in den Aktiv-Zustand wechselt, dann geht der PCA in den Zustand „Detectors Configured“ über, falls alle anderen Detektoren sich ebenfalls im Aktiv-Zustand befinden.
Die Teilbäume, welche QA und FLES beinhalten, müssen zusätzlich den Zustand betrachten, in welchem diese Daten aufzeichnen. Der PCA wechselt selbst in den aufzeichnenden Zustand, sobald sowohl FLES als auch QA im entsprechenden Zustand sind. Sobald einer von beiden aus dem aufzeichnenden Zustand heraus wechselt, geht auch der PCA wieder aus diesem Zustand heraus.
Ein Subsystem kann in einen Zustand wechseln, welcher weder aktiv, konfigurierend noch aufzeichnend ist. Der PCA muss in einem solchen Fall in einen Zustand wechseln, in welchem das Subsystem neu konfiguriert werden muss, falls der PCA sich nicht bereits in einem solchen Zustand befindet. Falls bei einem Subsystem eine solche Zustandsänderung auftreten sollte, wird eine für das Subsystem entsprechende Fehler-Transition ausgelöst, falls diese im aktuellen Zustand des PCA möglich ist. Wenn sich ein PCA zum Beispiel im Zustand „QA Configured“ befindet und ein Detektor in einen Fehlerzustand wechselt, wird die Transition „error_Detector“ ausgelöst und der PCA geht in den Zustand „TFC Configured“ über. Für den Fall, dass ein Detektor-Fehler im Zustand „TFC Configured“ auftritt, würde nichts geschehen, da der Detektor in diesem Zustand ohnehin neu konfiguriert werden müsste.
Das Betrachten aller möglichen Ereignisse von Subsystem- oder PCA-Zustandsänderungen in bestimmten globalen Zuständen spielt eine wichtige Rolle, da es ansonsten zu Deadlocks im PCA kommen kann, sodass dieser zum Beispiel in einem konfigurierenden Zustand festsitzt und keine weiteren Befehle annimmt.
An das Prüfen des aktuellen globalen Zustands sind neben der Transition der globalen *State Machine* noch weitere Aktionen gebunden.
Über ein Flag, welches beim Konfigurationsbefehl mitgesendet wird, lässt sich festlegen, ob nach einer erfolgreichen Konfiguration einer Ebene automatisch die Konfiguration der Subsysteme der nächsten Ebene gestartet werden soll.

Falls FLES oder QA sich im aufzeichnenden Zustand befinden und eine Fehlertransition auftritt, werden diese entsprechend aufgefordert, die Datenaufzeichnung zu beenden. Gleiches geschieht auch, falls FLES oder QA aus dem aufzeichnenden Zustand heraus wechseln.

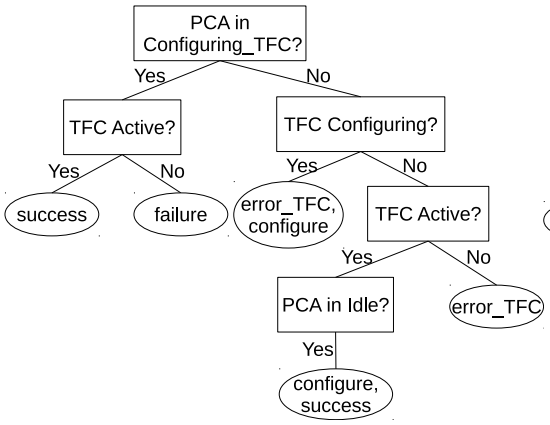

Abbildung 3.8: Teilbaum für TFC

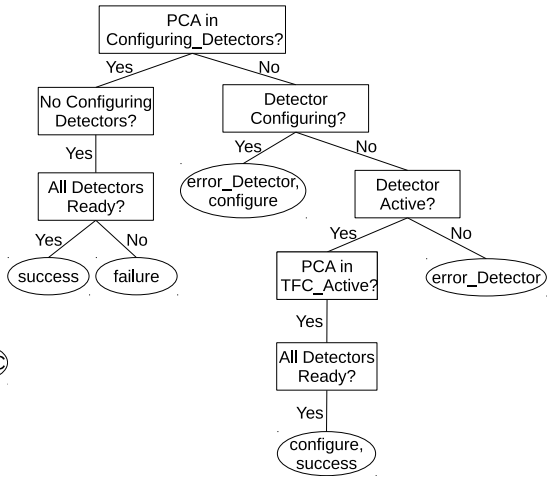

Abbildung 3.9: Teilbaum für Detektoren

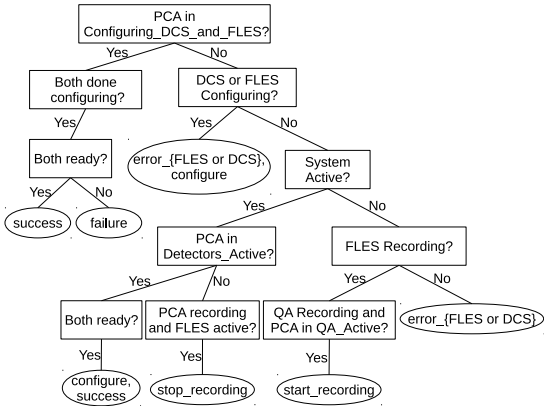

Abbildung 3.10: Teilbaum für FLES und DCS

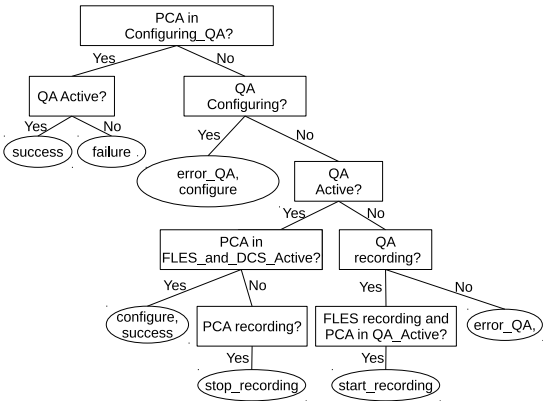

Abbildung 3.11: Teilbaum für QA

3.3 *Controller Agent*

Der *Controller Agent* dient als Schnittstelle zu einem Subsystem, um dieses zu überwachen und zu steuern. Durch ihn soll erreicht werden, dass die ECS-Logik des Subsystems nicht direkt im Subsystem selbst implementiert

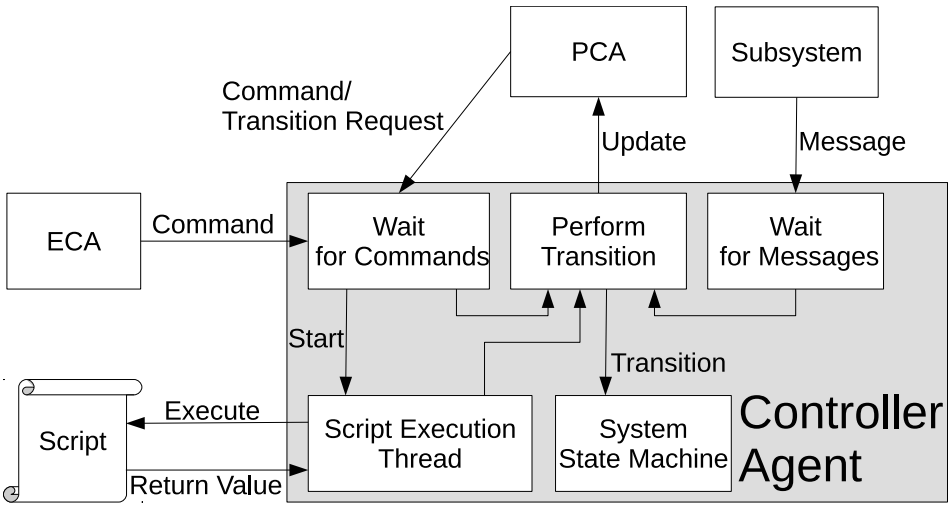

Abbildung 3.12: Der Aufbau eines *Controller Agents* mit seinen internen Komponenten und wie diese mit externen ECS-Komponenten in Verbindung stehen.

werden muss, sondern in einem eigenen Prozess läuft. Die *Controller Agents* verwalten ihren Systemzustand in *Finite State Machines*. Zustandsänderungen des überwachten Subsystems werden den zuständigen PCAs mitgeteilt. In Abbildung 3.12 sieht man den Aufbau eines *Controller Agents*.
Controller Agents existieren sowohl für Detektoren als auch für globale Systeme. Sie werden auf dem selben Knoten wie das zu überwachende System ausgeführt. In der Abbildung 3.13 sieht man das Klassendiagramm für *Controller Agents*. Wie auch bei den *Partition Component Objects* wird hier eine Vererbungshierarchie genutzt. Die allgemeinste Funktionalität befindet sich in einer Basisklasse, welche Controller-Funktionen sowohl für Detektoren als auch für globale Systeme bereitstellt. Von der Basisklasse wird jeweils eine allgemeine Klasse für Detektoren und globale Systeme abgeleitet. Ein *Controller Agent* leitet sich dann je nach Subsystem von einer dieser beiden Klassen ab. Der größte Unterschied zwischen einem Controller für Detektoren und einem Controller für globale Systeme ist, dass ein globales System mit allen Partitionen interagieren muss, während ein Detektor nur mit einer Partition interagiert. Die genaue Implementierung eines Subsystem-Controllers kann sich wie beim *Partition Component Object* von Subsystem zu Subsystem unterscheiden.

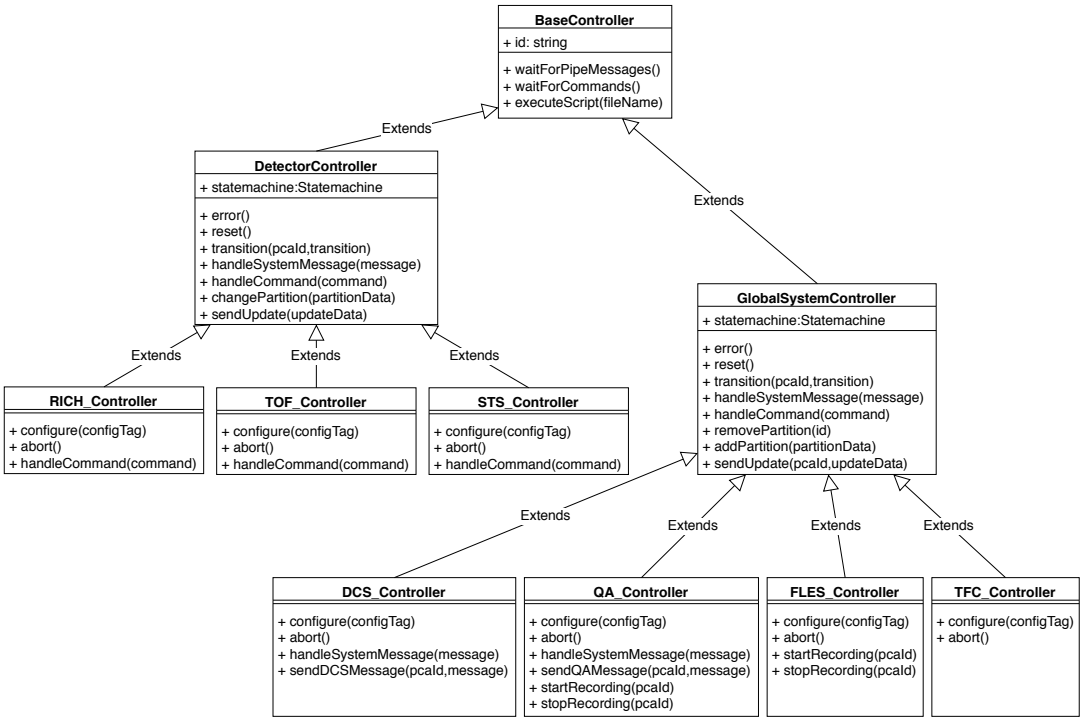

Abbildung 3.13: Klassendiagramm für *Controller Agents*

Die Zuordnung der entsprechenden Klasse des *Controller Agents* geschieht über das Typen-Attribut, welches für den Detektor in der Datenbank festgelegt wurde.
Die *Controller Agents* und deren *State Machines* agieren unabhängig voneinander. Falls bei einem *Controller Agent* ein Fehler auftritt, sind die Zustände von anderen *Controller Agents* hiervon zunächst nicht betroffen. Eventuelle Zustandsabhängigkeiten von Subsystemen werden auf der Partitionsebene gelöst.

3.3.1 Verwaltung von Kommandos und Subsystem-Nachrichten

Der *Controller Agent* wartet in einem Thread („Wait for Commands“ in Abbildung 3.12) auf eingehende Kommandos von einem PCA oder dem ECA. Bei Kommandos kann es sich um Transitions-Anfragen oder um generelle

Kommandos handeln. Generelle Kommandos können zum Beispiel Pings sein, Anfragen nach dem aktuellen Zustand der *State Machine* oder die Aufforderung den PCA zu wechseln, falls sich die Detektorzuweisung geändert hat. Transitions-Anfragen werden nur von PCAs versendet. Im Falle einer solchen Anfrage führt der Agent eine entsprechende Funktion aus, zum Beispiel zum Start der Konfigurierung des Subsystems. In diesen Funktionen wird dann ein neuer Thread gestartet („Script Execution Thread“ in Abbildung 3.12). Dieser Thread startet dann ein entsprechendes Shell-Script, über welches der Detektor oder das globale System gesteuert wird. Der *Controller Agent* wartet auf den Rückgabewert des Scriptes und bewertet diesen entweder als Fehler oder als Erfolg. Es kann immer nur eine Transition und daher auch nur ein Thread zum Ausführen von Scripten gleichzeitig ausgeführt werden. Falls der *Controller Agent* eine weitere Transition-Anfrage erhält, während eine andere nicht beendet ist, wird diese verworfen und dem PCA mitgeteilt, dass der *Controller Agent* noch beschäftigt ist. Ausgenommen hiervon ist eine Abbruch-Transition, welche zum Beispiel eine laufende Konfiguration abbrechen kann. Es besteht die Möglichkeit, dass ein Subsystem seinem *Controller Agent* Informationen mitteilen muss. Um dies zu ermöglichen, existiert ein weiterer Thread („Wait for Messages“ in Abbildung 3.12), welcher auf ankommende Nachrichten einer Pipe wartet. Über diese Pipe kann das zu dem Agent gehörende Subsystem Nachrichten senden. Nachrichten können unerwartete Fehlermeldungen sein oder zum Beispiel Monitoring-Nachrichten des DCS. Meldungen vom DCS oder QA, welche Detektoren oder ankommende Daten betreffen, werden an den PCA weitergeleitet, welcher dann notwendige Entscheidungen aufgrund dieser Meldungen trifft.
Alle genannten Threads können eine Transition der *State Machine* auslösen. Hierfür wird eine Methode („Perform Transition“ in Abbildung 3.12) aufgerufen. Diese Methode hat die Aufgabe, die *State Machine* in den neuen Zustand zu überführen und zuständige PCAs über die Zustandsänderung zu informieren. Bei der Überführung der *State Machine* handelt es sich um eine kritische Aktion, bei welcher die Möglichkeit besteht, dass sie von mehreren Threads gleichzeitig ausgeführt wird. Um Race-Conditions vorzubeugen, ist diese Methode durch ein Lock-Objekt abgesichert. Dadurch kann immer nur ein Thread die *State Machine* in neue Zustände überführen.

3.3.2 Eigenschaften für bestimmte Subsysteme

Der Unterschied zwischen einem *Detector Controller* und einem *Global System Controller* ist, dass ersterer nur eine *State Machine* für seinen Systemzustand benutzt. *Global System Controller* hingegen verwalten eine *State Machine* pro Partition. Dies ist notwendig, da sich ein globales System für mehrere Partitionen in verschiedenen Zuständen befinden kann, zum Beispiel, wenn im FLES für eine Partition nicht mehr genügend Rechenressourcen vorhanden sind.
Die globalen Systeme QA und DCS haben zusätzlich die Möglichkeit Meldungen an einen PCA zu schicken, bei welchen es sich nicht um Zustandsänderungen handelt. Im Falle vom DCS könnte dies zum Beispiel eine Meldung über einen Detektor sein, welcher nicht mehr genügend elektrische Spannung bekommt. Fehlermeldungen von QA und DCS, welche Detektoren oder ankommende Daten betreffen, haben keinen Einfluss auf den Zustand der eigenen *State Machine* von QA und DCS. Eine Fehlertransition bei diesen Systemen würde bedeuten, dass zum Beispiel das DCS selbst ausgefallen ist und nicht, dass das DCS einen Fehler bei einem Detektor bemerkt hat.

3.3.3 *Finite State Machines* für *Controller Agents*

In Abbildung 3.14 sieht man die *State Machine* des Prototyps, welche für das TFC, DCS und die Detektoren verwendet wird. Wie bei der globalen PCA-FSM wird zwischen Transitionen unterschieden, welche vom Benutzer und welche durch Ereignisse ausgelöst werden. Vom Benutzer induzierte Transitionen sind in Abbildung 3.14 durch fettgedruckte Schrift hervorgehoben.
Die FSM besteht aus vier Zuständen. Im Zustand „Unconfigured“ ist das System unkonfiguriert und daher nicht einsatzbereit. Über die „configure“-Transition gelangt man in den Zustand „Configuring“, welcher entweder mit einem Erfolgs- oder einem Fehlschlags-Ereignis endet. Bei einem Erfolg geht die FSM in den Zustand „Active“ über, in welchem das Subsystem aktiv ist. Bei einem Misserfolg wechselt das Subsystem zurück in den Zustand „Unconfigured“. Mit einem „abort“-Übergang kann man das aktive System oder

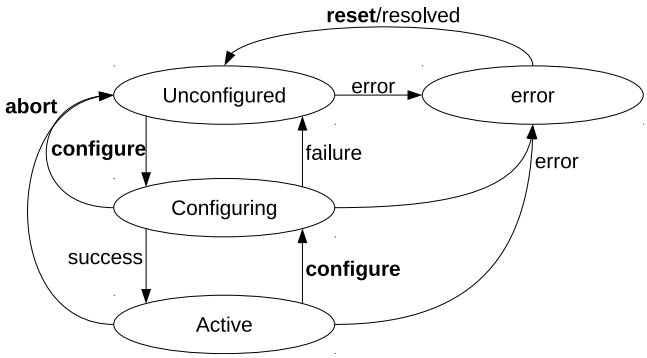

Abbildung 3.14: FSM für Detektor, TFC und DCS

die Konfiguration abbrechen. Im aktiven Zustand kann das Subsystem über eine „configure"-Transition direkt zurück in den konfigurierenden Zustand gelangen. Dieser Übergang existiert für den Fall, dass das Subsystem mit einem neuen Konfigurations-Tag neu konfiguriert werden soll. Da bei dem Subsystem eine Vielzahl von unerwarteten Fehlern auftreten kann, existiert zusätzlich ein Fehlerzustand namens „error". Um aus diesem Zustand zurück in den „Unconfigured"-Zustand zu kommen, muss entweder vom Detektor ein Ereignis ausgelöst werden, dass der Fehler nicht mehr vorhanden ist, oder der Detektor muss von einem Benutzer manuell zurückgesetzt werden.

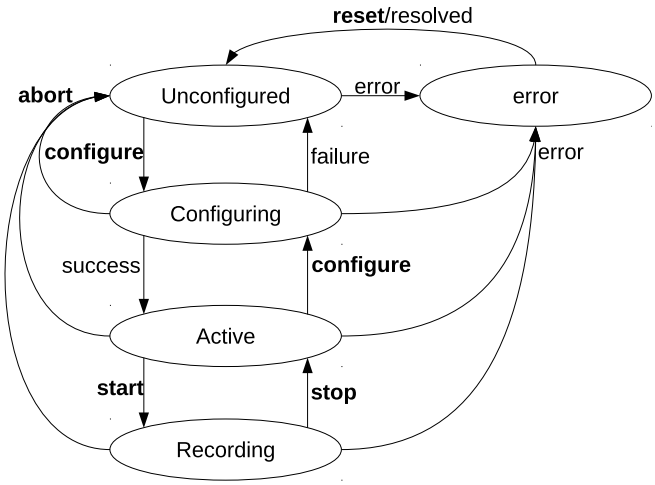

Abbildung 3.15: FSM für FLES und QA

Abbildung 3.15 zeigt die *State Machine* für FLES und QA. Bei dieser *State Machine* existiert zusätzlich der Zustand „Recording". Ansonsten ist die FSM identisch zu der der anderen Subsysteme. Der Unterschied zwischen den

Zuständen „Active“ und „Recording“ ist, dass im letzteren Daten vom Subsystem nicht nur Aktiv verarbeitet werden, sondern diese auch auf Festplatten gesichert werden.
Im Großen und Ganzen wird bei den Subsystemen nur zwischen einem Zustand unterschieden, in welchem das System aktiv ist, und einem, in welchem das System inaktiv ist. Das bedeutet, dass ein Subsystem anfängt, Daten zu nehmen und weiterzuleiten unabhängig davon, ob andere Systeme bereit sind oder nicht. Dadurch wird eine größere Unabhängigkeit zwischen den Systemen geschaffen, da Detektoren oder andere Subsysteme aktiv sein können, ohne zwangsläufig zu erfordern, dass alle anderen Systeme ebenfalls aktiv sind.
Wie auch bei den PCAs wird bei den *Controller Agents* der Übergangsgraph der *State Machine* aus einer entsprechenden CSV-Datei generiert.

3.4 Kommunikation zwischen ECS-Komponenten

Bei dem ECS mit seinen einzelnen Komponenten handelt es sich um ein System, welches über ein Netzwerk verteilt betrieben wird. Da die einzelnen Komponenten während des Betriebs Nachrichten untereinander austauschen müssen, wird an dieser Stelle ein Kommunikationsschema benötigt.
Die Kommunikation zwischen den einzelnen ECS-Komponenten erfolgt über ZeroMQ (*Zero Message Queue*). Bei ZeroMQ handelt es sich um eine Bibliothek zum Austausch von Nachrichten. Es bietet unter anderem Schnelligkeit, Asynchrone I/O, selbstständige Behandlung von auftretenden Netzwerkfehlern und eine API zu vielen gängigen Programmiersprachen, unter anderem auch Python.[10]
ZeroMQ kennt mehrere Patterns zum Ablauf der Kommunikation. Vom ECS wird das Request/Reply-Pattern und das Publisher/Subscriber-Pattern benutzt.
Beim Request/Reply-Pattern können mehrere Clients Anfragen an einen Server schicken, welcher diese dann beantwortet. Im Falle von mehreren Anfragen sorgt ZeroMQ für eine faire Verteilung. Im ECS sind diese Anfragen in

der Regel Updates oder Kommandos, welche mit einer Empfangsbestätigung beantwortet werden.
Mit dem Publisher/Subscriber-Pattern kann ein Publisher Nachrichten an mehrere Subscriber verteilen, wobei die Anzahl der Subscriber dynamisch skalierbar ist [10]. Dieses Pattern wird für Log-Nachrichten und Status-Updates von den PCAs benutzt.
Der Großteil der Kommunikation erfolgt zwischen den einzelnen Hierarchieebenen (siehe Abbildung 2.1), also zwischen ECA und PCA sowie zwischen PCA und *Controller Agents*.

3.4.1 Verwaltung von Kommunikationsadressen und Ports

ECA, PCAs und *Controller Agents* können alle auf unterschiedlichen Rechnern laufen, welche über ein Netzwerk verbunden sind, wodurch es sich um ein verteiltes System handelt. Ein Problem, welches dabei entsteht ist, dass alle Komponenten die IP-Adressen von allen Kommunikationspartnern benötigen. Auch die verwendeten Ports für die ZMQ-Sockets können variieren. Eine Möglichkeit wäre, die Kontaktdaten an jeden Knoten zu replizieren. Da sich die Kontaktdaten ändern könnten, wäre dies unpraktisch, weil dann alle Knoten synchronisiert werden müssten. Außerdem werden nicht von jedem Knoten alle Daten benötigt. Daher wurde eine zentrale Verwaltung der Kontaktdaten gewählt, welche über den ECA läuft. Alle anderen Knoten benötigen deshalb nur die Adresse und Ports des ECA.
Während des Startvorgangs fragen sowohl PCA als auch *Controller Agent* beim ECA nach Verbindungsdaten. Diese Daten sind in der Datenbank des ECS abgelegt (siehe Abbildung 3.2). Ein PCA benötigt die Portnummern für seine eigenen ZMQ-Sockets, Adresse und Ports für die globalen Systeme und eine Liste mit seinen zugewiesenen Detektoren ebenfalls mit Adresse und Ports. *Controller Agents* für Detektoren benötigen die eigenen Portnummern sowie die Adresse und Ports des PCA, welchem der Detektor zugewiesen ist. Agents für globale Systeme benötigen ebenfalls die eigenen Portnummern und zusätzlich Kontaktdaten für alle PCAs, da diese Teil aller Partitionen sind. DCS und QA benötigen außerdem die Zuweisungslisten für die Detektoren, um

beim Bemerken von Fehlern oder Auffälligkeiten zu wissen, welchem PCA dies gemeldet werden muss.

3.4.2 Kommunikationsaufbau für *Controller Agents*

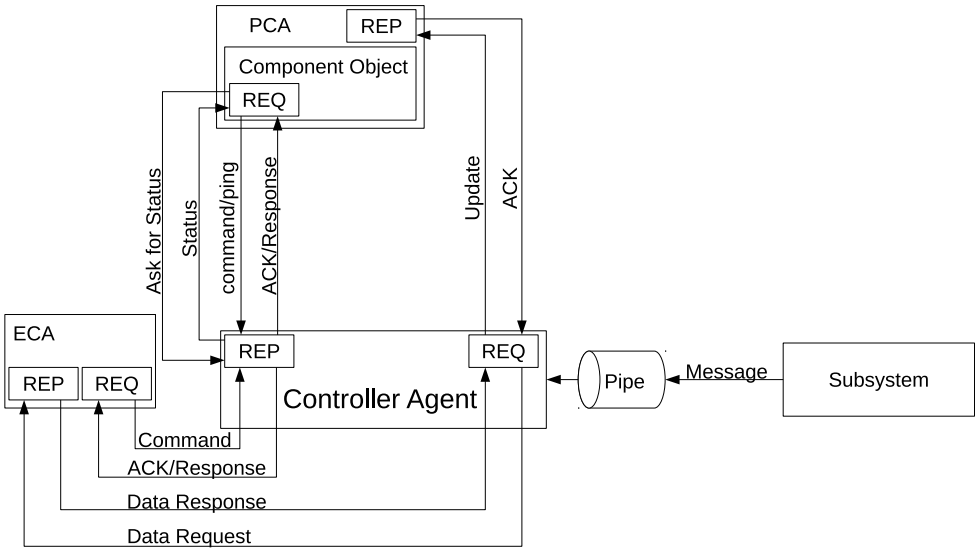

Abbildung 3.16: Kommunikationsaufbau für einen *Controller Agent*

In Abbildung 3.16 sieht man den Kommunikationsaufbau eines *Controller Agents* für ein Subsystem. Dieser besitzt einen Reply-Socket und erzeugt für Anfragen zu anderen Komponenten jeweils einen Request-Socket, welcher nach dem Ende der Anfrage wieder geschlossen wird.
Zwischen ECA und *Controller Agent* ist zum einen die bereits erwähnte Anfrage nach Verbindungsdaten aufgezeigt. Zum anderen kann der ECA direkt Kommandos an den Request-Socket des *Controller Agents* versenden, welcher den Empfang des Kommandos bestätigt oder mit einem Rückgabewert antwortet. Ein Kommando vom ECA kann beispielsweise die Aufforderung sein, den PCA zu wechseln, falls der Detektor verschoben wurde.
Im Regelfall erhält der *Controller Agent* die meisten Kommandos von seinem zugewiesenen PCA. Ein Kommando kann hier ein Ping oder eine Transitions-Anfrage sein. Der *Controller Agent* antwortet dann mit einer Empfangsbestätigung. Außerdem kann der PCA nach dem aktuellen Status des Subsystems fragen, worauf dann mit dem entsprechenden Status geantwortet wird.

Zuletzt kann der *Controller Agent* noch Nachrichten direkt von seinem Subsystem erhalten. Da der Agent auf demselben Knoten wie das Subsystem läuft, geschieht dies über eine *Named Pipe*. Nachrichten, die hier versendet werden, können Statusmeldungen über aufgetretene Fehler sein oder im Falle des DCS Monitoring-Nachrichten über einen Detektor.
Zustandsänderungen werden über einen Request-Socket an den PCA geschickt oder, falls der Detektor keiner Partition zugewiesen ist, an den *Unmapped Detector Controller* des ECA. Es ist wichtig, dass alle Updates vom PCA korrekt empfangen werden, da sonst innerhalb des Systems Inkonsistenzen bezüglich Subsystemzuständen von FSM und Statustabelle des PCA entstehen. Um diesem vorzubeugen, werden alle Status-Updates vom Empfänger mit einer Bestätigung beantwortet. Der Empfang der Bestätigung unterliegt einem Timeout mit konfigurierbarer Länge. Falls ein Timeout auftritt, versucht der *Controller Agent* die Nachricht solange erneut zu senden, bis er entweder eine Bestätigung erhält oder ein neueres Update gesendet werden muss.
Eine Nachricht über ein Status-Update besteht aus folgenden Teilen:

- Id des Detektors oder globalen Systems
- Sequenznummer
- Zustand der *State Machine*
- Konfigurations-Tag (falls vorhanden)
- Anmerkung (falls vorhanden)

Die Id wird benötigt, damit der PCA weiß, von wem die Nachricht kam, da alle Updates auf demselben Socket ankommen (siehe Abbildung 3.16).
Innerhalb des *Controller Agents* können Zustandsänderungen und deren Status-Nachrichten durch mehrere parallele Threads ausgelöst werden. Falls zwei Zustandsänderungen zum Beispiel sehr schnell hintereinander ausgelöst werden, ist nicht garantiert, dass die gesendeten Status-Updates in der Reihenfolge ankommen, in welcher sie ausgelöst wurden. Hierdurch können ebenfalls Inkonsistenzen zwischen Zuständen der Subsystem FSM und Zuständen innerhalb der Statustabelle des PCA entstehen. Dieses Problem kann durch das Verwenden von Sequenznummern, welche bei jedem versendeten

Update um eins erhöht werden, gelöst werden. Der PCA merkt sich die zuletzt erhaltene Sequenznummer und kann so ältere Updates von neueren unterscheiden. Updates mit einer Sequenznummer, welche kleiner als die Aktuelle ist, werden verworfen. Beim Start beziehungsweise Neustart versendet der *Controller Agent* das erste Update mit der Sequenznummer null. Der PCA weiß dann, dass es sich um das erste Update handelt und setzt seinen eigenen Sequenzzähler zurück. Falls eine Sequenznummer größer ist als die zuletzt empfangene, wird der neue Status in der Statustabelle des PCA abgelegt.
Das Konfigurations-Tag ist nur in konfigurierten oder konfigurierenden Zuständen vorhanden und existiert daher nicht immer.
Bei Anmerkungen handelt es sich um Kommentare zu dem aktuellen Zustand, welche dem Benutzer im Webinterface angezeigt werden; zum Beispiel, dass das Konfigurieren aus einem bestimmten Grund fehlgeschlagen ist oder vom Benutzer abgebrochen wurde.

3.4.3 Kommunikation zwischen ECA und PCAs

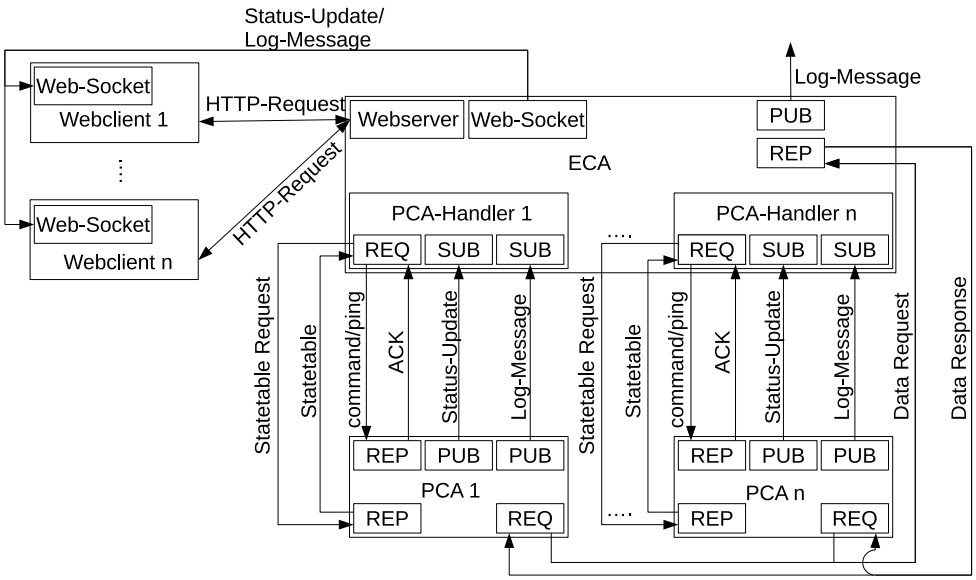

Abbildung 3.17: Kommunikationsaufbau von ECA und PCAs

In Abbildung 3.17 sieht man den Kommunikationsaufbau von ECA und PCAs. Wie auch die *Controller Agents* erfragen sich die PCAs beim Start die Detektor-

zuweisung und Verbindungsdaten vom ECA.
Für jeden PCA-Handler werden zwei Subscription-Sockets erstellt, einer zum Empfangen von Zustandsänderungen des PCA, und ein zweiter zum Empfangen von Log-Nachrichten des PCA. Der ECA sammelt alle Log-Nachrichten der PCA-Handler und veröffentlicht diese zusammen mit eigenen Log-Nachrichten auf einem Publisher-Socket, mit welchem sich dritte Programme verbinden können. Für jedes Kommando an einen PCA wird vom entsprechenden PCA-Handler ein Request-Socket erstellt, welcher Nachrichten an den Reply-Socket des entsprechenden PCA sendet und vom diesem eine Empfangsbestätigung bekommt. Das Warten auf diese Bestätigungen unterliegt Timeouts, deren Länge sich über eine Konfigurationsdatei festlegen lässt. Im Falle eines Timeouts wird dieser im Log festgehalten, welches auch für Benutzer des *Web User Interfaces* sichtbar ist.
Wie bei den Updates, welche vom *Controller Agent* zum PCA geschickt werden, weist der PCA den Updates eine Sequenznummer zu. Diese Sequenznummer entspricht nicht der des *Controller Agents*, sondern einer unabhängigen Zählung des PCA, welche bei jeder globalen oder Subsystem-Zustandsänderung um eins erhöht wird. Diese Sequenznummer wird zusammen mit den vom *Controller Agent* erhaltenen Statusinformationen und mit dem gemappten Zustand aus PCA-Sicht in der Statustabelle des PCA abgelegt.
Bei einer neuen Subscription kann das *Late Joiner Problem* entstehen. Dieses Problem entsteht, wenn sich ein Subscriber erst nach dem Start des Publisher mit diesem verbindet. Der Subscriber hat in diesem Fall eventuell vergangene Updates verpasst. Zur Lösung des Problems existiert ein Reply-Socket, auf welchem man die aktuelle Statustabelle des PCA abfragen kann. Jeder Subscriber muss nach einer Subscription einen Snapshot der aktuellen Statustabelle abfragen. Das Empfangen von Snapshots und neuen Updates laufen parallel zueinander ab. Einträge des Snapshots und neue Updates werden in eine lokale Statustabelle eingetragen. Falls für einen Key in der Statustabelle ein Konflikt auftritt, wird das Update mit der niedrigeren Sequenznummer verworfen. Auf diese Weise erhält der Subscriber eine synchronisierte Statustabelle.
Beim Start beziehungsweise Neustart des PCAs veröffentlicht dieser ein Reset-Signal an seine Subscriber. Beim Erhalt dieses Signals verwerfen diese ihre

aktuelle Tabelle und erfragen einen neuen Snapshot der aktuellen Tabelle. Selbiges geschieht auch, falls ein Verbindungsabbruch zum PCA bemerkt wird.

3.4.4 Kommunikation zwischen ECA und Clients

Status-Updates werden vom ECA durch einen Websocket weiter veröffentlicht. Browserclients verbinden sich mithilfe von Javascript mit diesen Sockets und fügen die Daten, inklusive der Sequenznummern, mithilfe von jQuery an die entsprechende Stelle in den HTML-Code ein. Auch hier existiert das *Late Joiner Problem*, welches hier analog gelöst wird. Javascript wartet, bis die Verbindung offen ist und erfragt sich dann über eine HTTP-Anfrage eine Kopie der aktuellen Statustabelle von dem Webserver. Hiernach werden wieder die Sequenznummern von Kopie und neuen empfangenen Updates verglichen.
Beim Öffnen der Verbindung der Websockets bekommt der ECA mitgeteilt, welche Partitions-Updates der neu geöffnete Websocket erhalten möchte. Wenn sich ein Benutzer beispielsweise auf der Übersichtsseite einer bestimmten Partition befindet, benötigt dieser nur Updates für diese bestimmte Partition. Updates werden dann nur an die Sockets versendet, welche von diesen betroffen sind, wodurch unnötiger Daten-Traffic vermieden wird.
Zusätzlich veröffentlicht der ECA Log-Nachrichten auf einem ZMQ-Publisher-Port, auf welchem sich weitere – auch Nicht-Webclients – verbinden können. Auf diesem Port werden sowohl Log-Nachrichten vom ECA selbst als auch von dessen PCA-Handlern empfangene Logs veröffentlicht.
Das Versenden von Kommandos vom Web-Client und Anfragen auf Webseiten laufen beide über HTTP-Anfragen an den Webserver.

3.5 Webserver und Nutzeroberfläche

Die Nutzeroberfläche ist webbasiert und läuft über einen Webserver. Das Benutzen einer webbasierten Oberfläche besitzt die Vorteile, dass das Interface zum Gebrauch nur einen Webbrowser und ansonsten keine weitere Software

benötigt, sodass es weitgehendst plattformunabhängig ist und dass auf Seite des Clients kein Wartungsaufwand durch Versionsaktualisierung entsteht.
Zum Erstellen des Webinterfaces wird das Django-Framework verwendet. Bei Django handelt es sich um Web-Framework zum Erstellen von Webseiten in der Programmiersprache Python [11]. Der ausschlaggebende Grund für die Wahl dieses Framework ist, dass das ECS ebenfalls in Python implementiert ist, und es daher leicht zu integrieren ist.
Die Webseiten werden vom Web-Client per HTTP-Anfrage vom Server abgefragt. Um die Webseiten möglichst dynamisch zu gestalten, kommt die Javascript-Bibliothek jQuery und AJAX zum Einsatz, über welche Daten zum oder vom Server übertragen werden können ohne die Seite erneut zu laden. Auf den Übersichtsseiten der Partitionen und des ECS werden Websockets und jQuery verwendet, um Updates dynamisch auf der Webseite einzufügen, ohne dass der Benutzer diese manuell aktualisieren muss.
Zum Gestalten der Oberfläche wird das CSS-Framework Bootstrap genutzt zusammen mit dem Template „SB-Admin“[1], welches als Vorlage dient. Dieses Framework und das Template wurden ausgewählt, weil durch diese eine Vielzahl von bewährten und getesteten Funktionen bereitgestellt wird. Hierdurch muss zum einen das Rad nicht neu erfunden werden und zum anderen wird späterer Wartungsaufwand an der Oberfläche verringert.
Django wird mit einem eigenen Webserver ausgeliefert, welcher allerdings nur zur Entwicklung der Webseite benutzt werden sollte. Für ein Deployment sollten Webserver wie Apache oder Nginx verwendet werden, welche das Deployment von Django-Web-Applikationen unterstützen und häufig zu diesem Zweck genutzt werden [11].
Zum Verwenden der Webseite werden Benutzerzugänge benötigt. Daher werden vom Webclient aus Benutzernamen und Passwörter übertragen. Um diese zu sichern, muss der Server so konfiguriert sein, dass dieser nur über SSL verschlüsselte Verbindungen über das HTTPS-Protokoll zulässt. Auch die Verbindungen des Websockets können und sollten über SSL verschlüsselt werden, was in einer Konfigurationsdatei entsprechend festgelegt werden kann.

3.5.1 Berechtigungen

Es kann jeweils nur ein Benutzer eine kontrollierende Rolle für einen PCA einnehmen. Die Kontrolle der PCAs durch mehrere Benutzer gleichzeitig ist nicht sinnvoll, da deren Befehle miteinander konkurrieren. Zudem ist die Verwaltung von parallelen Kommandos aufwändiger zu lösen. Alle anderen Benutzer, welche keine kontrollierende Rolle besitzen, nehmen eine beobachtende Rolle ein, in welcher sie nur den aktuellen Status und Log-Nachrichten betrachten können.
Die Verwaltung der Berechtigungen läuft über den Webserver. Um das *User Interface* zu benutzen, wird ein Login-Account benötigt. Neue Accounts können von einem Administrator über eine von Django bereitgestellte Admin-Seite angelegt werden. Dabei kann entschieden werden, ob der neu angelegte Benutzer generell nur eine beobachtende Rolle haben soll oder auch die Möglichkeit haben soll, eine kontrollierende Rolle einzunehmen.
Für jede Partition kann unabhängig voneinander jeweils ein Benutzer eine kontrollierende Rolle einnehmen. Falls ein anderer Benutzer die Kontrolle über einen PCA übernommen hat, bekommen alle anderen Benutzer dies angezeigt. Damit ein anderer Benutzer die Kontrolle übernehmen kann, muss der alte Besitzer erst explizit seine Rolle abgeben. Für den Fall, dass jemand vergisst, die Kontrolle abzugeben, besteht die Möglichkeit, einen Timeout nach zu langer Inaktivität festzulegen, nach welchem alle eingenommenen Berechtigungen dieses Benutzers wieder freigegeben werden. Hiermit soll vermieden werden, dass niemand mehr eine Partition steuern kann, weil ein anderer Benutzer vergessen hat, seine Berechtigung abzugeben. Auch beim Abmelden des Benutzers werden dessen eingenommene Rechte wieder freigegeben.
Auch für das ECS als Ganzes gibt es eine kontrollierende Rolle, welche jeweils nur ein Benutzer innehaben kann. Diese wird benötigt, um neue Detektoren, Partitionen oder Subsystem-Konfigurationen anzulegen oder Detektoren zwischen Partitionen zu verschieben.
Die Berechtigungen werden über das Django-Framework umgesetzt, welches die Account-Daten in einer eigenen Datenbank ablegt. Für jeden PCA und für das gesamte ECS wird vom Server ein Objekt in der Django-Datenbank

angelegt, über welches sich der Server die aktuellen Rechte an der Partition beziehungsweise ECS merkt. Da Django generell nur Berichtigungen für Datenbank-Tabellen kennt, wird an dieser Stelle die Library „django-guardian“ verwendet, welche auch Berechtigung auf einzelnen Datensätzen innerhalb einer Tabelle unterstützt. Im Falle einer unautorisierten Anfrage antwortet der Server mit einem Statuscode 403 (Forbidden).
Die Websockets laufen über eine Python-Bibliothek namens „websockets“. Diese Bibliothek ist kein Bestandteil des Django-Frameworks und hat somit keinen direkten Zugriff auf dessen Authentifizierungs-Funktionalitäten. Um dieses Problem zu lösen, ist im Webserver ein Signal-Handler für das Ein- und Ausloggen des Benutzers implementiert. Beim Einloggen wird dem Benutzer in der Django-Datenbank seine Session-Id zugeordnet, auf welche der Websocket zugreifen kann. Beim Ausloggen wird diese Zuordnung wieder aufgehoben. Nach dem Öffnen der Verbindung muss der Web-Client seinen Benutzernamen und seine Session-Id senden. Auf Serverseite wird dann die Gültigkeit der Daten geprüft. Im Falle der Gültigkeit wird die Verbindung zum Versenden von Statusnachrichten freigegeben. Ansonsten wird die Verbindung vom Server beendet. Der Empfang der Authentifizierungsdaten unterliegt auf Seite des Servers einem einstellbaren Timeout. Falls dieser eintreten sollte, wird die Verbindung ebenfalls geschlossen.

3.5.2 Nutzeroberfläche

Die Abbildung 3.18 zeigt einen Screenshot der PCA-Übersichtsseite. Alle Seiten der Nutzeroberfläche werden oben und links von einem Rahmen umgeben. Oben Rechts im Rahmen kann man Benutzeraktionen auswählen wie zum Beispiel Ein- und Ausloggen oder die Kontrolle über Partitionen einnehmen oder aufgeben. Der linke Rand der Oberfläche dient zur Navigation zu den Übersichtsseiten von Partitionen oder Funktionsseiten. Eine Funktionsseite ist zum Beispiel die Seite zur Änderung von Detektorzuweisungen. Der Rest der Anzeigefläche wird für den eigentlichen Seiteninhalt verwendet.
Alle Webseiten der Nutzeroberfläche verwenden die Vererbung der *Django Template Language*, indem sie von einer Basisseite erben. Dadurch entsteht zum

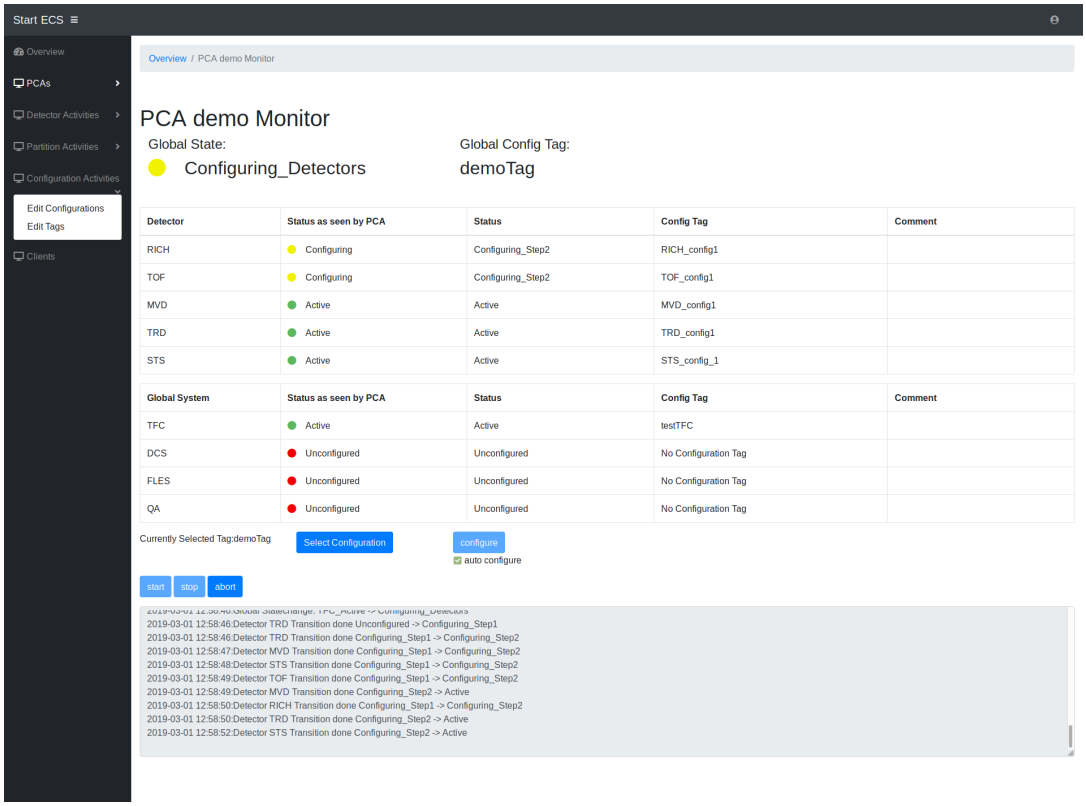

Abbildung 3.18: Screenshot des *Web User Interfaces*

einen der Vorteil, dass für das Einbinden und Entwickeln weiterer Webseiten nur bestimmte Blöcke innerhalb dieser Basisseite überschrieben werden müssen und zum anderen, dass für globale Änderungen nur die Basisseite geändert werden muss.
Benutzungselemente oder Links zu anderen Webseiten, für welche ein Benutzer keine Berechtigung hat, werden ausgeblendet und erst angezeigt, sobald der Benutzer die entsprechend benötigten Rechte eingenommen hat.
Auf den Übersichtsseiten für PCAs und ECS werden Log-Nachrichten und Statusupdates über Websockets empfangen. Empfangene Log-Nachrichten werden in eine HTML-Textbox eingefügt. Zustände werden in einer Tabelle angezeigt.
Insgesamt kann man mit der Nutzeroberfläche sowohl Partitions-Verwaltungsaufgaben ausführen, wie zum Beispiel das Verschieben von Detektoren als auch Partitionen steuern und betrachten. Hierdurch lässt sich der gesamte Funktionsumfang des ECS über die Nutzeroberfläche nutzen.

4 Messungen und Tests

Das System wurde auf dem Mini-FLES-Cluster an der GSI aufgesetzt und die Funktionalität des ECS wurde erfolgreich getestet. Da für die meisten Subsysteme noch keine Konfigurations-Scripte und Interfaces zum ECS vorhanden sind, wurden als Stellvertreter einfache Bash-Scripte verwendet. Diese Scripte führen ein Sleep-Kommando mit vorgegebener Länge aus und geben dann einen Rückgabewert zurück. Die Pipes der *Controller Agents* wurden direkt über Bash-Terminals angesprochen. Als Webserver wurde während des Testens der Entwicklungsserver des Django-Frameworks verwendet.

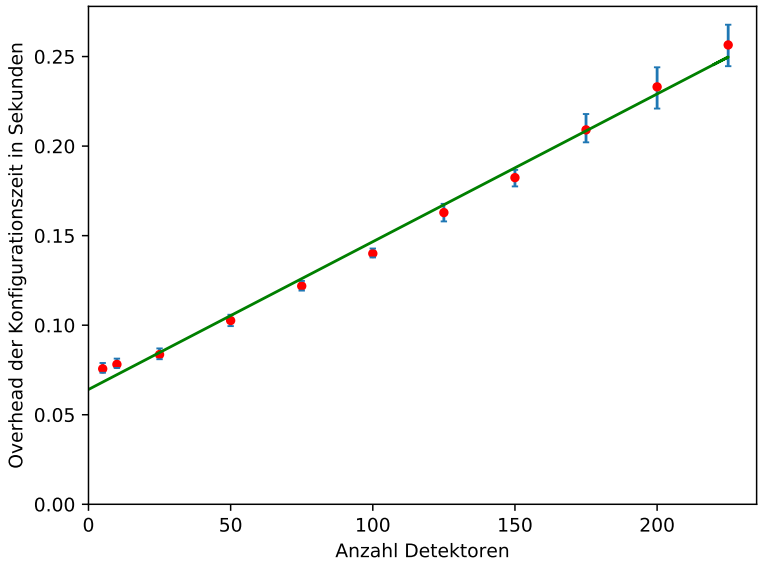

Abbildung 4.1: Messwerte für den Overhead des Systems. Die roten Punkte zeigen die Mittelwerte für die jeweiligen Zeitmessungen. Die blauen Balken zeigen die minimale und maximale gemessene Abweichung. Die grüne Gerade ist eine lineare Kurvenanpassung an die gemessenen Mittelwerte.

Abbildung 4.1 zeigt die Werte zur Messung des Overheads für eine vollständige Konfiguration des Systems. Gemessen wurde hierbei die Länge vom Eingang eines Befehls zum Konfigurieren bis zur Bereitschaft des gesamten Systems. Entsprechend der *State Machine* des PCA (siehe Abbildung 3.6) bedeutet dies die Dauer vom Zustand „Idle“ bis „QA-Configured“. Im Bash-Script zum Konfigurieren der Subsysteme wurde eine Wartezeit von zwei Sekunden bis zum Rückgabewert festgelegt. In der benutzten *State Machine* des Detektors wurden zwei Konfigurationsschritte verwendet, wobei in jedem Schritt das Bash-Script ausgeführt wird. Daraus ergibt sich für die Detektoren eine ideale Konfigurationszeit von vier Sekunden. Alle globalen Systeme führen das Script nur einmal aus. Da alle Detektoren sowie auch das DCS und FLES parallel konfiguriert werden, erhält man einen Idealwert von zehn Sekunden zur vollständigen Konfiguration. Um zu sehen, wie gut das System bei steigender Detektor-Anzahl skaliert, wurde die Messung mit größeren Detektor-Mengen wiederholt. Um zu simulieren, dass die ECS-Komponenten auf unterschiedlichen Rechnerknoten laufen, wurden diese über den *Slurm Workload Manager* im Mini-FLES-Cluster verteilt.
Das Ergebnis der Zeitmessungen sieht man in Abbildung 4.1. Die y-Achse zeigt die Mittelwerte der benötigten Zeit (abzüglich der fixen zehn Sekunden) zur Konfigurierung des gesamten Systems und die x-Achse die Anzahl der Detektoren. Die Zeitmessung wurde pro Anzahl Detektoren 15 mal ausgeführt. Die roten Punkte zeigen jeweils den Mittelwert der Messungen für eine Anzahl von Detektoren. Die blauen Balken zeigen die höchste und niedrigste Abweichung vom Mittelwert. Die grüne Linie ist eine an die gemessenen Mittelwerte angepasste Gerade.
Man kann auf dem Graphen eine lineare Steigung der benötigten Zeit für eine steigende Detektor-Anzahl sehen, wobei auch bei 200 Detektoren der Overhead unter 250 Millisekunden bleibt. Der lineare Anstieg entsteht dadurch, dass bei den Detektoren die Konfiguration zwar parallel gestartet wird, aber die Verarbeitung von ankommenden Zustandsänderungen – wie bereits im Kapitel 3 beschrieben – sequentiell erfolgt.
Vor dem Hintergrund, dass im CBM-Experiment je nach Konfiguration eine Detektor-Anzahl von sechs oder sieben vorgesehen ist[3] und das Konfigurieren durchaus eine größere Zeit in Anspruch nehmen kann, sind die erhaltenen

Werte zufriedenstellend.
Für den FLES existiert bereits ein Start-Script, welches den FLES als Batch-Job über Slurm auf dem Cluster startet. Ein Problem dabei ist, dass zurzeit – wie auch bei anderen Subsystemen – kein Interface zum ECS existiert und konfigurierende, aktive und datennehmende Zustände schlecht unterscheidbar sind. Die *Controller Agents* verlassen sich nur auf die Rückgabewerte der Bash-Scripte. Im Fall von FLES ist dies die *Slurm-Job-Id* des Batch-Jobs. Der FLES-Controller wurde dahingehend angepasst, dass dieser sich die Job-Id speichert und den Status des Jobs beobachtet. FLES wird als konfiguriert betrachtet, sobald der Job läuft. Bei einem Abbruch-Befehl wird der Slurm-Job abgebrochen, was vom Controller bemerkt wird, welcher daraufhin in den Unkonfiguriert-Zustand wechselt. Der datennehmende Zustand kann hierbei nicht abgebildet werden.
Durch diesen Test konnte gezeigt werden, dass das ECS in der Lage ist, eine komplette Datennahme erfolgreich zu starten. Durch den Versuchsaufbau in einem verteilten Netzwerk wurde gezeigt, dass das ECS auch als verteiltes System – mit teils heterogenen Subsystemen – funktioniert.
Die Django-Applikation und der ECA wurden auch über das *Web Server Gateway Interface* (WSGI) in einen Apache-Webserver eingebunden. Die Serveranfragen laufen hierbei über das HTTPS-Protokoll und auch die Websocket-Verbindung wird über SSL verschlüsselt. Auch in diesem Szenario wurde die Funktionalität des ECS erfolgreich getestet.

5 Diskussion

Das Ergebnis dieser Arbeit ist ein lauffähiger Prototyp, in welchem bereits ein großer Teil an Funktionalität implementiert ist. Mit diesem Prototyp wird ein konzeptioneller Beweis für den erdachten Aufbau eines ECS für ein triggerloses Experiment wie CBM erbracht. In der Implementierung wurde darauf geachtet den, Prototypen möglichst ausbaufähig zu erhalten, damit dieser gegebenenfalls in Zukunft als Grundlage für das finale ECS für CBM dienen kann.
Mit der erstellten Django-Webapplikation wird ebenfalls ein Prototyp für eine webbasierte Benutzeroberfläche geboten, welche bereits Funktionalitäten wie die Verwaltung von Benutzerrechten und das Anzeigen auf einem Websocket empfangener Daten implementiert. Die vorhandene Oberfläche kann ebenfalls gut als Grundgerüst für eine Erweiterung dienen.
Für eine Weiterentwicklung des Prototypen wird unter anderem eine Schnittstelle zwischen den *Controller Agents* und den entsprechenden Subsystemen benötigt. Diese Schnittstelle muss sowohl das Versenden von Datenanfragen und Befehlen vom Controller an das Subsystem implementieren als auch die Übertragung von Informationen vom Subsystem an den Controller. Ein Rückschluss auf den aktuellen Zustand eines Subsystems aufgrund eines Rückgabewertes eines Konfigurations-Scriptes ist nicht ausreichend, da zum Beispiel nach der Konfiguration ein Fehler während der Datennahme auftreten könnte, über welchen der *Controller Agent* informiert werden muss.
Eine wichtige Eigenschaft für das CBM-ECS ist, dass es keine Trigger-Hierarchien gibt und Subsysteme so prinzipiell unabhängiger voneinander sein können als zum Beispiel im ALICE-ECS. Diese Eigenschaft wird dadurch erfüllt, dass Subsysteme ihren Zustand wechseln ohne zwangsläufig andere Systeme zu beeinflussen. Falls ein Detektor in einen Fehlerzustand übergeht,

bleiben andere Detektoren in ihren aktuellen Zuständen und der fehlerhafte Detektor kann individuell neu konfiguriert werden. Für globale Systeme besteht zudem die Möglichkeit für unterschiedliche Partitionen in voneinander unabhängigen unterschiedlichen Zuständen zu sein.
Aufbauend auf ZeroMQ wurde ein Kommunikationssystem geschaffen, über welches Zustandsänderungen einer beliebigen Menge von Subscribern mitgeteilt werden können. Durch Bemerken von Verbindungsabbrüchen und erneutes Synchronisieren der Daten im Fehlerfall bietet dieses System ein gewisses Maß an Ausfallsicherheit. Über dieses System könnten im zukünftigen ECS auch andere Informationen als FSM-Zustände verteilt werden.
In dieser Arbeit wurden auch Vorschläge sowohl für eine globale *State Machine* für Partitionen (siehe Abbildung 3.6) als auch für *State Machines* für Subsysteme (siehe Abbildung 3.14 und 3.15) vorgestellt. Diese FSMs entsprechen wahrscheinlich nicht den finalen *State Machines*, wie sie in Zukunft zum Einsatz kommen. Da die FSMs im Programmcode mithilfe von CSV-Dateien initialisiert werden, muss bei einer Änderung einer *State Machine* am Code der FSM selbst nichts geändert werden. Die Logik, wann ein Übergang der *State Machine* stattfindet, kann an einigen Punkten jedoch benötigte Änderungen am Code herbeiführen. Besonders betroffen hiervon ist zum Beispiel die Methode des PCAs, welche nach einer Subsystem-Zustandsänderung den globalen Zustand bestimmen muss.
Insgesamt ist eine gute Grundlage für ein zukünftiges ECS für das CBM-Experiment entstanden.

6 Zusammenfassung und Fazit

Ein *Experiment Control System* wird in Teilchenbeschleuniger-Experimenten benötigt um alle beteiligten Detektoren und sonstigen Subsysteme des Experiments zu einem Gesamtsystem zu verbinden. Von triggerlosen Experimenten wie CBM werden neue Anforderungen an das ECS gestellt, vor allem in Hinblick auf Unabhängigkeit der verschiedenen Subsysteme.
Innerhalb dieser Masterarbeit wurde ein Prototyp für ein in dieser Form benötigtes ECS entwickelt. Die Struktur des Systems orientiert sich am ALICE-ECS und bietet ebenfalls die Möglichkeit der Detektor-Partitionierung, verzichtet allerdings auf Trigger-Hierarchien und benutzt ein einfaches Publisher/Subscriber-System über ZMQ-Sockets zur Veröffentlichung von Daten.
Das entwickelte ECS besteht aus mehreren Komponenten welche hierarchisch miteinander in Verbindung stehen. Der *Experiment Control Agent* verwaltet eine Menge von *Partition Control Agents*, welche eine Partition von *Controller Agents* verwalten, welche jeweils ein Subsystem verwalten. Detektoren werden genau einer Partition zugeordnet während globale Systeme wie der FLES Teil aller Partitionen sind.
Zur Erfassung von Systemzuständen werden *Finite State Machines* verwendet. Es existieren FSM für Subsysteme als auch globale FSM für Partitionen. Die *State Machines* für Subsysteme sind vollständig unabhängig von anderen Systemen. Für globale FSM muss der aktuelle Zustand aus den Zuständen der Subsysteme abgeleitet werden. Um das Ableiten des globalen Zustands leichter zu gestalten, werden die Zustandsmengen der Subsystem-FSMs auf globaler Ebene jeweils auf eine kleinere Zustandsmenge abgebildet. Bei den meisten Subsystemen wird im Großen und Ganzen nur zwischen einem Zustand unterschieden, in welchem das System aktiv ist, und einem Zustand, in welchem das System inaktiv ist.

Das ECS ist über eine Weboberfläche einsehbar und steuerbar. Die Oberfläche wird von einer Webapplikation bereitgestellt, welche im selben Prozess wie der ECA läuft. Zustandsänderungen von *State Machines* werden über Websockets an die Webclients übertragen. Über diese Webapplikation wird auch sichergestellt, dass eine Partition von Detektoren jeweils von nur einem Benutzer gleichzeitig gesteuert werden kann, wobei weitere Benutzer weiterhin den Systemzustand beobachtend verfolgen können.
Die einzelnen Komponenten des ECS sind unabhängige Prozesse, welche über ein Netzwerk verteilt auf mehreren Rechnern laufen können. Diese Komponenten sind mithilfe von ZeroMQ über ein komplexes Kommunikationsschema verbunden, welches auch in der Lage ist, Verbindungsabbrüche zu behandeln. Der entwickelte ECS-Prototyp wurde auf einem Cluster an der GSI erfolgreich getestet und bietet eine gute Grundlage für das zukünftige ECS des CBM-Experiments.

Literaturverzeichnis

[1] *SB Admin*. `https://startbootstrap.com/templates/sb-admin/`, März 2019.

[2] T Ablyazimov, A Abuhoza, R P Adak, M Adamczyk, K Agarwal, M M Aggarwal, Z Ahammed, F Ahmad, N Ahmad, S Ahmad, A Akindinov, P Akishin, E Akishina, T Akishina, V Akishina, A Akram, M Al-Turany et al.: *Challenges in QCD matter physics – The scientific programme of the Compressed Baryonic Matter experiment at FAIR*. The European Physical Journal A, 53, 03 2017.

[3] Tomas Balog: *Overview of the CBM detector system*. Journal of Physics: Conference Series, 503:012019, 04 2014.

[4] F Carena, W Carena, P Van de Vyvre, J Marin, S Chapeland, R Divià, A Vascotto, C Soos und K Schossmaier: *The ALICE experiment control system*. 2005.

[5] W Carena, P Van de Vyvre, F Carena, S Chapeland, V Chibante Barroso, F Costa, E Denes, R Divia, U Fuchs, G Simonetti, C Soos, A Telesca und B von Haller: *ALICE DAQ ECS User's Guide*. Technischer Bericht ALICE-INT-2005-015. CERN-ALICE-INT-2005-015, CERN, Geneva, 2005.

[6] ALICE Collaboration: *ALICE trigger data-acquisition high-level trigger and control system: Technical Design Report*. Technischer Bericht, CERN, 2004. Kapitel 16.

[7] CBM Collaboration: *Letter of Intent for the Compressed Baryonic Matter Experiment at the Future Accelerator Facility in Darmstadt*. `https://fair-center.eu/fileadmin/fair/experiments/CBM/documents/LOI2004.pdf`, 2004.

[8] J de Cuveland, V Lindenstruth und CBM Collaboration: *A First-level Event Selector for the CBM Experiment at FAIR*. Journal of Physics: Conference Series, 331(2):022006, 2011.

[9] C Gaspar und M Dönszelmann: *DIM – A distributed information management system for the DELPHI experiment at CERN*. Technischer Bericht, 1994.

[10] Pieter Hintjens: *ZeroMQ*. O'Reilly Media, 2013.

[11] Ayman Hourieh: *Django 1.0 Website Development*. Packt Publishing, 2009.

[12] Lukas Dominik Meder: *Timing Synchronization and Fast-Control for FPGA-based large-scale Readout and Processing Systems*. Doktorarbeit, Karlsruher Institut für Technologie (KIT), 2017.

[13] B Povh, K Rith, C Scholz und F Zetsche: *Teilchen und Kerne. Eine Einführung in die physikalischen Konzepte*, Kapitel Nukleare Thermodynamik. Springer-Verlag GmbH, neunte Auflage, 2014.