FH | JOANNEUM
University of Applied Sciences

Implementierung eines persistenten und skalierenden Job-Schedulers

Bachelorarbeit 2
zur Erlangung des akademischen Grades
Bachelor of Science in Engineering (BSc)
eingereicht am
Fachhochschul-Studiengang **Software Design**

FH JOANNEUM (University of Applied Sciences), Kapfenberg

Betreuer: FH-Prof. Dipl.-Ing. Dr. Egon Teiniker

eingereicht von: Mario Löfler
Personenkennzahl: 1310418017

Juni 2017

Zusammenfassung

Die wiederkehrende Erledigung von Aufgaben ist in der EDV ein häufig anzutreffendes Problem. Dabei müssen die Aufgaben in etwa zu einem bestimmten Zeitpunkt in der Zukunft erledigt werden oder wiederkehrend in vorgegebenen Zeitabständen ausgeführt werden - ohne Interaktion eines Benutzers.
Im Internet der Dinge werden solche Systeme auch oft eingesetzt um Daten von Geräten zu holen die diese nicht aktiv zur Verfügung stellen können. Im Internet der Dinge kann diese Anzahl dabei rasch hoch werden, das System muss mit der wachsenden Anzahl an Geräten in der Leistungsfähigkeit mitwachsen und die erwarteten Tätigkeiten ausführen.
In dieser Arbeit wird ein solches System das sowohl funktionell als auch in der Leistung skaliert entwickelt und prototypisch eingesetzt. Einige der wichtigen Themen werden auch in der Theorie erklärt und die Umsetzung im Code erläutert. Schwerpunkt dieser Erläuterungen sind insbesondere die Problemstellungen bei der Persistenz der durchzuführenden Aufgaben und der Sicherstellung der Leistung.
Die Umsetzung wird zur Lösung einer tatsächlichen Aufgabenstellung eingesetzt. Der Theorieteil basiert auf einer Literaturrecherche.
Zielpublikum der Arbeit sind Entwickler die an Serversystemen arbeiten die wiederkehrende Aufgeben zu erledigen haben oder ähnliche Problemstellungen im Bereich der Skalierung zu erfüllen haben.

Abstract

Recurring tasks are a common challenge in software systems. Tasks either need to be scheduled to be executed at a future point in time or need to be executed in set intervals without further user interaction.
The Internet of things often uses such systems to fetch data from devices that can not actively push the data. The number of devices within the Internet of things is often large and heterogeneous, so the tasks that need to be run are diverse. The system therefore needs to be able to scale in performance and functionality.
As a part of this thesis such a system is developed put to use in a real environment. Exemplary problems like persistence and concurrency are elaborated in this thesis. The theoretical part is backed by literature research. The target audience of this theses are developers that either face the challenge of recurring tasks or are generally interested in the scaling problems.

Ehrenwörtliche Erklärung

Ich erkläre ehrenwörtlich, dass ich die vorliegende Bachelorarbeit selbstständig angefertigt und die mit ihr verbundenen Tätigkeiten selbst erbracht habe. Ich erkläre weiters, dass ich keine anderen als die angegebenen Hilfsmittel benutzt habe. Alle aus gedruckten, ungedruckten oder dem Internet im Wortlaut oder im wesentlichen Inhalt übernommenen Formulierungen und Konzepte sind gemäß den Regeln für gutes wissenschaftliches Arbeiten zitiert und durch Fußnoten bzw. durch andere genaue Quellenangaben gekennzeichnet.

Die vorliegende Originalarbeit ist in dieser Form zur Erreichung eines akademischen Grades noch keiner anderen Hochschule vorgelegt worden. Diese Arbeit wurde in gedruckter und elektronischer Form abgegeben. Ich bestätige, dass der Inhalt der digitalen Version vollständig mit dem der gedruckten Version übereinstimmt.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

Graz, 21. Juni 2017 Mario Löfler

Inhaltsverzeichnis

Kapitel 1

Einleitung

In diesem Kapitel stelle ich das zugrundeliegende Problem einführend vor. Aus einer allgemeinen Beschreibung werden im Zusammenspiel mit einer beispielhaften Verwendung der zu entwickelnden Software die grundlegenden Anforderungen erarbeitet. Aus diesen werden die Ziele dieser Arbeit definiert.

1.1 Motivation

Die meisten Benutzer von Computerprogrammen nutzen diese interaktiv. Sie befinden sich in einem Dialog mit der Software wo sie aufgrund ihrer Eingabe eine bestimmte Ausgabe erwarten. Handelt es sich bei den Aufgaben die so dem Programm gestellt werden um länger laufende Aktionen, so gibt der Benutzer bloß den Anstoß, die Aufgabe selbst wird ohne weitere Interaktion erledigt und das Ergebnis angezeigt. Die meiste Zeit ist die Aufmerksamkeit des Benutzers nicht notwendig. Gibt es nun mehre Aufgabenstellungen dieser Art die auch noch wiederholt ausgeführt werden müssen, so sollten diese automatisiert werden, so dass sie ohne Eingriff des Benutzers ausgeführt werden und dieser nur mehr bei Problemen informiert wird.
Bei Aufgaben dieser Art handelt es sich oft auch um Datenimporte, bei denen aus unterschiedlichen Quellsystemen Daten geladen und in Zielsysteme importiert oder mit den Daten in Zielsystemen abgeglichen werden.
Die regelmäßige Ausführung von Programmen wird durch *Scheduler* übernommen. Dies sind Programme die gemäß einem durch den Benutzer definierten Zeitplan Aufgaben automatisiert ausführen. Zwei der bekanntesten Vertreter sind *cron* unter Linux oder der *Windows Task Scheduler* für Microsoft Windows.
Beide Scheduler bieten eine gute Möglichkeit Programme basierend auf Zeitplänen

auszuführen, beide haben jedoch die Einschränkungen dass sie

- Programme nur auf einem Rechner ausführen und
- komplette Programme ausführen.

Wenn die Anzahl der Aufgaben die ausgeführt werden soll ansteigt, müssen die Aufgaben durch den Benutzer selbst auf mehrere Rechner verteilt werden. Wenn ein Rechner ausfällt oder ein neuer Rechner hinzukommt muss der Benutzer die Aufgaben ebenso neu verteilen.
Ebenso sind beide Scheduler darauf ausgelegt komplette Programme, also *executables* auszuführen. Dadurch müssen diese für Ausführung ein neuer *Prozess* angelegt werden und die dazugehörigen Daten erneut in den Speicher geladen werden. Werden nun häufig die selben Aufgaben erledigt und sind diese relativ klein, so steht der Zusatzaufwand des Ausführens meist nicht im Verhältnis zur Ausführungszeit der Aufgabe selbst.
Für den Fall der umzusetzen war stellten diese beiden Herausforderung ein großes Hindernis dar. Nach der Anforderungsanalyse und einer Machbarkeitsstudie wurde daher entschieden einen eigenen Scheduler umzusetzen.

1.2 Beispielhafte Verwendung

Eines der Hauptthemen in der Industrie 4.0 ist die Einbindung von Sensoren in EDV Systeme um durch deren Daten einerseits Informationen zu aktuellen (Betriebs-) Zuständen zu erhalten und andererseits durch die Sammlung der Daten mithilfe von Big-Data Algorithmen weitere Schlussfolgerungen ziehen zu können.[MHS15, S. 36ff]
Dabei wird in der Regel stillschweigen von der neuesten Generation von Sensoren ausgegangen, die ihrerseits aktiv Daten in die Cloud publizieren.[1] Diese Sensoren sind aktiv und lösen somit die weitere Verarbeitung der Daten aus. Es ist keine Aktivierung von Außen notwendig.
Viele der derzeit bestehenden Sensoren sind allerdings passiv und stellen ihre Daten nur nach Nachfrage zur Verfügung. Die Daten dieser Sensoren müssen also periodisch abgefragt und an die weiterverarbeitenden Prozesse weitergereicht werden.
In der beispielhaften Verwendung geht es darum Standbilder von bereits bestehenden Überwachungskameras zu Dokumentationszwecken in einem Archiv abzulegen. Die Anzahl der Kameras beträgt in etwa 1.000 und wird mit der Zeit deutlich steigen, damit muss das System frei skalieren.

[1] Siehe zum Beispiel [Mic17] "...the device sends..."

1.3 Ziele der Arbeit

Ziel der Arbeit ist es einen funktionstüchtigen Scheduler der auf mehreren Rechnern läuft und nicht vollständige Prozesse ausführt zu erstellen. Im Rahmen dieser Arbeit wird nur der Scheduler Kern entwickelt, die Oberflächen zum Einrichten und Verwalten der Aufgaben, sowie das Einsehen der Protokolle wird nicht umgesetzt.
Im Theorieteil werden die zugrundeliegenden Themen im Bereich Datenbanken und Concurrency[2] mittels Literaturrecherche erläutert.
Durch diese Erläuterungen sollen die getroffenen Designentscheidungen untermauert werden und einem Entwickler die Themenkomplexe verständlich vermittelt werden.

1.4 Verwendete Programmierumgebung

Da eine der technischen Anforderungen die Ausführbarkeit unter Microsoft Windows ist, habe ich mich zur Umsetzung der Lösung mittels C# und dem Microsoft .Net Framework v 4.6 entschlossen, als Datenbank wird Microsoft SQL Sever 2016 verwendet. Die Entwicklung erfolgte unter Microsoft Visual Studio 2017. Als objektrationales Mapping Tool zur einfachen Anbindung von Datenbanken wurde das Microsoft Entity Framework 6.1 verwendet.
Die Entscheidung war durch folgende Askepte geprägt

- Lauffähigkeit unter Microsoft Windows
- Automatische Installierbarkeit von Sicherheitspatches für die Laufzeitumgebung
- Hohe Ausführungsgeschwindigkeit unter Microsoft Windows
- Mögliche Portierbarkeit auf andere Betriebssysteme mittels Microsoft .Net Core das zu diesem Zeitpunkt bereits in einer ersten Version zur Verfügung steht.

[2]Zu Deutsch Nebenläufigkeit. Im Sinne der besseren technischen Verständlichkeit werde ich im folgenden nur den englischen Fachbegriff verwenden.

Kapitel 2

Anforderungen

In diesem Kapitel stelle ich die Anforderungen an das System vor. Der Fokus liegt bei den für das Thema relevanten Anforderungen, es wird keine vollständige Anforderungsanalyse durchgeführt.

2.1 Allgemein

Wie schon Eingangs erwähnt, soll das System Bilder in regelmäßigen Abständen von Kameras abholen und diese auf einem Archivserver speichern. Die Abholung erfolgt dabei - je nach Standort der Kamera - einmal alle Stunden oder einmal jede Minute.
Da die Kameras getauscht werden und neue Kameras hinzukommen ist es notwendig, dass die Einrichtung der einzelnen Aufgaben für den Benutzer einfach und auch während des Betriebes erfolgen kann.
Die im folgenden angeführten Anforderungen sind vom System auf alle Fälle zu erfüllen. Die Beschreibung der Anforderungen folgt dem Schema von User-Stories.

2.2 Funktionelle Anforderungen

Im folgenden sind die wichtigsten funktionellen Anforderungen an den Scheduler angeführt:

1. Das System kann Aufgaben ausführen die aus einer Aktion (Aufgabentyp) und einer zu der Aktion gehörenden Konfiguration gehören.

2. Das System kann beliebig viele Aufgaben verwalten und diese bestmöglich zu den geforderten Zeitpunkten ausführen. Die Anzahl der Aufgaben sollte zumindest 100.000 sein.

3. Als Benutzer kann ich für jede Aufgabe ein oder mehrere Zeitpläne festlegen in dem die Aufgabe abgearbeitet wird. Der Zeitplan muss folgende Einteilungen zulassen

   (a) Einmal zu einem festgelegtem Zeitpunkt

   (b) Periodisch alle x Sekunden

   (c) Stündlich

   (d) Täglich zu einer festgelegten Uhrzeit

   (e) Wöchentlich an einem festgelegtem Tag und einer festgelegten Uhrzeit

4. Das System muss verschiedene Aufgaben unterstützen. Der Scheduler muss ohne Änderungen in der Lage sein verschiedene Aufgabentypen zu erledigen.

5. Das System muss nicht echtzeitfähig sein, eine Abweichung der tatsächlichen Ausführungszeitpunkten gegenüber den vorgegebenen Zeitpunkten ist erlaubt. Die Aufgaben sollten jedoch mit einer möglichst geringen Abweichung zu den definierten Zeitpunkten abgearbeitet werden.

6. Das System stellt die Ergebnisse der Ausführung dem Benutzer als einsehbares Log zur Verfügung.

7. Das System kann Aufgaben auf mehreren Rechnern ausführen. Die Verteilung erfolgt dabei automatisch durch den Scheduler. Neue Rechner können dem Scheduler ohne Änderung der Konfiguration hinzugefügt werden.

8. Ein Ausfall eines Rechners wird durch das System automatisch erkannt und behoben. Die Ausführung von Aufgaben wird dann automatisch auf alle andern Rechner verteilt. Die auf dem Rechner zum Zeitpunkt der Deaktivierung bearbeiteten Aufgaben sollen bestmöglich wiederhergestellt und erneut ausgeführt werden. Aufgaben die nicht wiederhergestellt werden können .

9. Einzelne Rechner sollen nur bestimmte Aufgabentypen ausführen. Dadurch können für die jeweiligen Typen spezialisierte Rechner verwendet werden.

2.3 Technische Anforderungen

Im folgenden sind die wichtigsten technischen Anforderungen an den Scheduler angeführt:

1. Der Scheduler muss unter Microsoft Windows Server lauffähig sein.
2. Die Aufgaben sind als Klassen definiert.
3. Die auf einem Rechner zur Verfügung stehenden Klassen müssen ohne Anpassung des Scheduler Codes oder der Konfiguration änderbar sein.
4. Der Scheduler kann mehrere Aufgaben pro Rechner gleichzeitig verarbeiten. Eine Synchronisation gegenüber den Ressourcen die von den Aufgaben benötigt werden ist nicht in der Verantwortung des Schedulers.
5. Der Administrator kann die Anzahl der Prozessoren die pro Rechner für den Scheduler verwendet werden konfigurieren.

Kapitel 3

Grundlagen

In diesem Kapitel werden die theoretischen Grundlagen zu den wichtigsten Bereichen der zu entwickelnden Software erläutert. Auf eine Einführung in die verwendete Programmiersprache und deren Ökosystem wurde aus Platzgründen verzichtet. Es werden ausgewählte Themen aus den Bereichen relationale Datenbank und Concurrency beschrieben.

3.1 Relationale Datenbanken

asda

3.1.1 Modellierung

dasds

3.1.2 Primary Key

dfsdf

3.1.3 Locks und Isolation

asdas

3.2 Concurreny

asdas

3.2.1 Threads

asdas

3.2.2 Synchronisierung

dasdas

Kapitel 4

Umsetzung

In diesem Kapitel wird die Umsetzung der Anforderungen in die Software erläutert. Es werden dabei besonders die Aspekte der Datenpersistenz, der Concurrency und der Skalierbarkeit vorgestellt.

4.1 Überblick und Architektur

asdasd

4.2 Datenbank

asdasd

4.2.1 Datenbankmodell

aasdas

4.2.2 Zugriff

sdasdaas

4.3 Skalierung

sdfsfs

4.3.1 Skalierung der Leistung

sdsds

4.3.2 Funktionale Skalierung

sdsd

```
foreach ( string file in  workerDirectory.GetFiles( "*.dll" ) )
{
        try
        {
                Assembly assembly = Assembly.LoadFile( file );
                logger.LogDebug( "[REPO]    scanning '{0}'", file );

                foreach ( Type typeToLoad in assembly.GetTypes()
                                                .Where( t => t.ImplementsInterface<IWorker>() ) )
                {
                        logger.LogDebug( "[REPO] loading '{0}'", typeToLoad.FullName );

                        IWorker w = (IWorker)Activator.CreateInstance( typeToLoad );

                        w.Initialize( ObjectRepository.Get<IConfigurationProvider>( ), logger );

                        if( ! repository.TryAdd( typeToLoad.FullName, w ) )
                        {
                                throw new InvalidOperationException
                                ( $"Could not add class '{typeToLoad.FullName}' to repository");
                        }
                }
        }
        catch ( Exception x )
        {
                logger.LogError( "[REPO] Error loading file '{0}': {1}"
                                , file
                                , x.GetDetailString( ) );
                sb.AppendLine( $"Error loading file '{file}': {x.GetDetailString()}");

        ++errorCount;
        }
}
```

Listing 4.1: Dynamic Loading

4.4 Concurrency

dasdasd

4.4.1 Threadpool

asda

4.4.2 Producer-Consumer

adadsa

4.4.3 Blocking-Queue

Listing BlockingQueue-V1

```
1           /// <summary>
2       /// own, experimental, implementation of ablocking, notifying Q for a producer/ consumer model
3       /// </summary>
4       public class MyBlockingQueueV1<T>
5       {
6               /// <summary>
7               /// The backing store used for the elements
8               /// </summary>
9               private Queue<T> backingStore = new Queue<T>();
10
11              /// <summary>
12              /// lock to protect the q
13              /// </summary>
14              private object myLock = new object();
15
16
17
18              public MyBlockingQueueV1()
19              { /* empty */ }
20
21              public void Enque( T newElement )
22              {
23                      lock ( myLock )
24                      {
25                              backingStore.Enqueue( newElement );
26                      }
27              }
28
29
30              public T Dequeue()
31              {
32                      lock ( myLock )
33                      {
34                              return backingStore.Dequeue();
35                      }
36              }
37
38              public T TryDequeue()
39              {
40                      lock( myLock )
41                      {
42                              if( !IsEmpty() )
43                              {
44                                      return Dequeue();
45                              }
46                              else
47                              {
48                                      return default( T );
49                              }
50                      }
51              }
52
53              public bool IsEmpty()
```

```
54                  {
55                          lock ( myLock )
56                          {
57                                  return backingStore.Count == 0;
58                          }
59                  }
60          }
```

4.5 Verbesserungsmöglichkeiten

Kapitel 5

Konklusio

5.1 Verwendbarkeit der erstellten Software

dsds

5.2 Abschlussbemerkung

asdasd

Abbildungsverzeichnis

Listings

Literatur

Manzei, Christian, Ronald Heinze und Linus Schleupner (2015). *Industrie 4.0 im internationalen Kontext - Kernkonzepte, Ergebnisse, Trends*. Düsseldorf: Beuth. ISBN: 978-3-800-73671-3.

Microsoft (2017). *.NET Documentation - AppDomain.AssemblyResolve-Ereignis*. Available from: <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-identity-registry> [15. Aug. 2017].