Aufgabe 1 - Modellierung

Zu erwartendes Lastaufkommen

Bei der Berechnung des erwarteten Lastaufkommens in der Vorweihnachtszeit gehen wir vereinfacht davon aus, dass es auf der Welt etwa 5 Milliarden Kinder und kindgebliebene Erwachsene gibt, die ihre Wunschliste an Santa Claus übermitteln möchten. Außerdem gehen wir davon aus, dass all diese Einsendungen innerhalb von 90 Tagen (im letzten Quartal) eingehen und jede Person nur *einen* Wunschzettel, wenn auch mit mehreren Wünschen, einsendet.

Dabei wurde vereinfachend die Annahme gemacht, dass die gesamte Menschheit den Weihnachtsmann als religions- und kulturunabhängige Instanz respektiert und eine fristgerechte Lieferung der Geschenke an dem sich historisch in einer seit Jahrtausenden fortlaufenden Studie als am günstigsten herausgestellten Liefertermin (dem 24.12.) gewohnt ist. Maßgeblich bei der Berechnung des Liefertermins ist das zu diesem Zeitpunkt vorliegende Gleichgewicht zwischen Gemütlichkeit des Abends (nimmt mit Fortschreiten des Winters zu) und Wahrscheinlichkeit einer schneesturmfreien Nacht (nimmt mit Fortschreiten des Winters ab). Dabei wird entgegen der Realität davon ausgegangen, dass der Jahreszeitenzyklus überall auf der Erde synchron verläuft.

Wie dem auch sei, das System durchläuft jedes Jahr drei Phasen, die je ein wachsendes Anfrageaufkommen im *XMasWishes*-System erwarten lassen:

- **Phase 1: Wunschlisten-Einsammel-Phase** - 90 Tage vor dem Stichtag wird das System auf öffentlich geschaltet und ist bereit für eingehende Wunschlisten. Dabei wird innerhalb dieser Zeit etwa 5 Milliarden mal schreibend auf das System zugegriffen. Die Anfragefrequenz in Hertz berechnet sich wie folgt:

$$freq_{collection} = 5000000000 \div (90 \cdot 24 \cdot 60 \cdot 60)hz \approx 650hz$$

- **Phase 2: Produktionsphase** - In den letzten 30 Tagen vor dem Stichtag produzieren die Elfen alle Geschenke. Hierbei fragt jeder Elf aus dem System einen Wunsch ab, der noch nicht bearbeitet wurde, um diesen dann umzusetzen. Der Wunsch wird dann auf *In Bearbeitung* gesetzt. Hier wird jeder Wunsch genau ein mal abgefragt. Bei durchschnittlich 3 Wünschen pro Wunschzettel ergibt sich das Anfrageaufkommen wie folgt:

$$freq_{production} = 3 \cdot 5000000000 \div (30 \cdot 24 \cdot 60 \cdot 60)hz \approx 5800hz$$

- **Phase 3:Auslieferungsphase** - Am Stichtag lädt der Weihnachtsmann alle Geschenke auf seinen Schlitten und setzt den Zustand aller Geschenke von *In Bearbeitung* auf *In Zustellung*. Dies ist mit einem einzelnen Aufruf möglich. Danach ruft er während der Auslieferung jede Wunschliste noch einmal ab, um in jedem Haus die passenden Geschenke abzuliefern. Wem unrealistisch erscheint, dass der Weihnachtsmann alle Geschenke selbst ausliefert, dem sei gesagt, dass der Weihnachtsmann sich mithilfe modernster Quantentechnologie stets in einer Superposition befindet und somit an mehreren Orten gleichzeitig sein kann. Es wird also innerhalb eines Tages jede Wunschliste einmal in einem Aufruf abgerufen und die entsprechenden Geschenke danach als ausgeliefert markiert. Das Anfrageaufkommen ergibt sich dann wie folgt:

$$freq_{delivery} = 2 \cdot 5000000000 \div (1 \cdot 24 \cdot 60 \cdot 60)hz \approx 120000hz$$

Modellierung einer skalierbaren Architektur für XMasWishes

Folgender Ansatz wird in diesem Projekt umgesetzt:

- **Backend:** Horizontal skalierbare, verteilte Datenbank.
- **Business-Logik:** Stateless Service, der die Schnittstelle zwischen Nutzer und Datenbank herstellt. Kann beliebig oft repliziert werden.
- **Load-Balancer** Wird zwischen Frontend und Business-Logik geschaltet, um die Last gleichmäßig auf die Services zu verteilen.
- **Frontend** Web-Applikation im Browser des Nutzers, ggf. auf dem gleichen Server wie die Business-Logik gehostet.

Ein Authentifizierungsservice wie OAuth wird nicht implementiert, da der Nordpol bereits über eine quantenbasierte Firewall verfügt, die sicherstellt, dass jeder nur auf das zugreifen kann, was er benötigt.

Aufgabe 2 - Konkretisierung

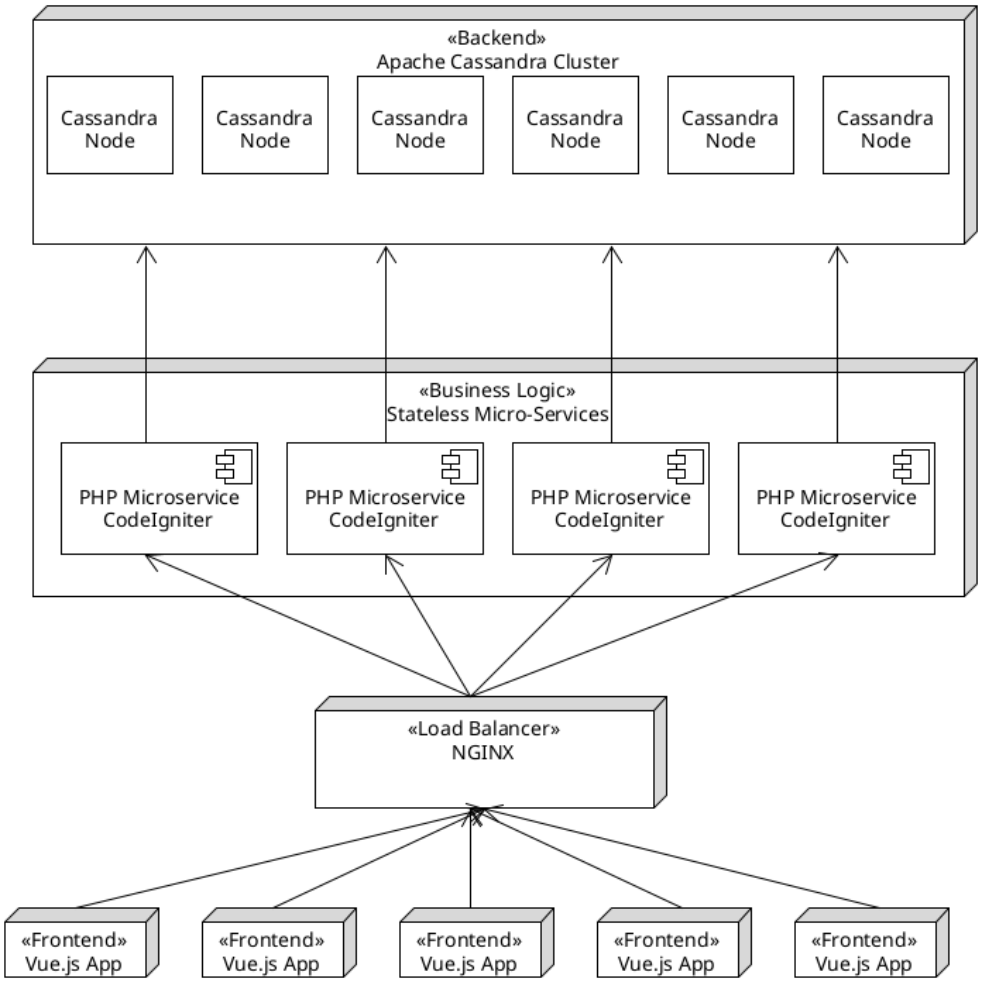

Abbildung 1: Architektur des Systems

Ich habe mich bei der Implementierung für folgende Technologien entschieden:

- **Backend:** Als Backend-Datenbank habe ich mich für **Apache Cassandra** entschieden. Dabei handelt es sich um eine horizontal skalierbare, verteilte NoSQL-Datenbank, die eine relationale Datenbank simuliert und mit einer SQL-Ähnlichen Sprache angesprochen wird. Sie skaliert in Performance und Speicherkapazität linear mit der Anzahl der Knoten und ist somit ideal für große Datenmengen und großes Transaktionsaufkommen geeignet. Mithilfe von Hashing wird bereits durch den Datenbank-Driver des Clients (in diesem Fall der Geschäftslogik) festgestellt, auf welchem Datenbank-Node eine Anfrage am schnellsten bearbeitet werden kann. So wird die Last optimal auf das Cluster verteilt.
- **Business-Logik:** Bei der Business-Logik setze ich auf **RESTful Microservices**, bzw. in diesem Fall nur einen WishlistService, der die Wunschlisten und Wünsche verwaltet. Weitere Microservices wie eine Nutzerverwaltung und einen Authentifizierungs-Service spare ich mir an dieser Stelle, könnten aber einfach hinzugefügt werden. Diese Services sind stateless (speichern keine Sitzungsdaten), und können daher einfach repliziert werden. Als Technologie habe ich mich für **PHP** mit dem Framework **CodeIgniter** entschieden. Vorteil daran ist, dass der Service auf einem normalen Apache-Webserver läuft, der sehr leichtgewichtig ist. CodeIgniter ist ebenfalls ein Framework, dass sehr auf Performance und Leichtgewicht optimiert ist, und nur die nötigsten Funktionen zur Verfügung stellt, so zum Beispiel eingebautes Routing, Separation nach dem MVC-Modell, diverse eingebaute Datenbank-Treiber sowie einige essenzielle Sicherheits-Features. Durch das Hosting auf einem Apache-Server und die Verwendung von CodeIgniter könnte außerdem die Verteilung von Frontend-Websites auch über diesen Service realisiert werden.
- **Load-Balancer:** Um die Last gleichmäßig auf die Instanzen des WishlistService zu verteilen, wird **NGINX** als Load-Balancer verwendet. NGINX ist ein hochperformanter Webserver, Reverse-Proxy und Load-Balancer, der in diesem Fall die Anfragen nach dem Round-Robin-Prinzip auf die vorhandenen Services verteilt. Er erkennt außerdem Ausfälle einzelner Service-Instanzen und leitet Anfragen ggf. um.
- **Frontend:** Das Frontend wird als **Vue.js Single-Page-Applikation** realisiert. Dadurch wird die gesamte App mit dem ersten Aufruf des Web-Links geladen und dann lokal beim Nutzer gecached. So wird die Last des Webservers reduziert. Um die Last des Webservers weiter zu reduzieren, kann auch dieser repliziert und hinter einen Load Balancer geschaltet werden. Außerdem wäre es möglich, Bilder, CSS- und Javascript-Dateien über ein *Content Delivery Network (CDN)* bereitzustellen, um so weiter die Last des Webservers zu reduzieren. Dieser müsste dann nur noch eine wenige Zeilen lange HTML-Datei ausliefern.

EDIT: Änderungen wegen Problemen in Aufgabe 3

- Ich hätte wirklich gerne PHP verwendet für die Implementierung der Business-Logik, um zu sehen, wie sich aktuelle PHP-Versionen so schlagen von der Performance her. Leider ist mir nach ewigen herumprobieren aufgefallen, dass der PHP-Support des Cassandra-Treibers für PHP bereits bei PHP 7.1 geendet ist. Da ich nicht mit seit Jahren veralteter Technologie arbeiten wollte, habe ich mich daher wohl oder übel dazu entschieden, den **Microservice mithilfe von Node.js und Typescript** umzusetzen.

Aufgabe 3 - Prototyp

Um ein verteiltes System zu simulieren, verwende ich **Docker-Compose** zur Verwaltung einiger Docker-Container, die die entsprechenden Maschinen in der Architektur emulieren. So ist es auch möglich, die Lösung schnell und einfach auf einem anderen Rechner laufen zu lassen.

Bei der Implementierung habe ich einen Back-to-Front-Ansatz verfolgt. Ich habe also damit begonnen, die Datenbank einzurichten, dann die Microservices erstellt, den Load-Balancer konfiguriert und zuletzt die Frontend-Applikation gebaut.

Für die Datenbank habe ich ein Cassandra-Cluster mit drei Knoten erstellt. Sobald das Cluster erfolgreich hochgefahren ist, startet ein weiterer Cassandra-Container, der mithilfe von `cqlsh` eine CQL-Datei auf der Datenbank ausführt, um den Keyspace und die benötigten Tabellen einzurichten. Dies ist nur beim ersten Hochfahren des Clusters relevant, da die Daten in den Containern der Nodes persistent gespeichert sind.

Als nächstes ging es dann an die Entwicklung des Wishlist-Services. Leider wurde ich hier dann doch mit den Limitierungen bzw. Eigenheiten von Cassandra konfrontiert: Anders als bei einer Relationalen Datenbank ist das Suchen nach Werten durch das Schlüsselwort `WHERE` nur auf dem Primärschlüssel einer Tabelle effizient möglich. Daher ist zum Beispiel ein Statement, dass alle Geschenke, die auf *In Bearbeitung* stehen, auf *In Zustellung* stellt, nicht so einfach umzusetzen und erst recht nicht effizient. Um die dadurch entstehende Lastspitze beim Beladen des Schlittens abzufangen die Logistik etwas geändert, sodass die Geschenke nun, sobald sie produziert sind, von dem produzierenden Elfen auf den Schlitten gelegt werden, der dann den Wunsch entsprechend kennzeichnet. Aber vielleicht wäre es auch egal durch die reine Anzahl an Cassandra-Nodes, die im Produktivbetrieb ja nötig wären. Da kenne ich mich leider nicht genug aus mit Cassandra, um da ein fundiertes Urteil zu fällen.

Das Einrichten von NGINX als Load Balancer ging sehr einfach und bedurfte nur weniger Zeilen Konfiguration in der `default.conf`. Ich habe beispielhaft drei Instanzen des Wishlist-Service eingebaut und den Load Balancer entsprechend konfiguriert. Zum Testen habe ich einen Ping-Endpoint im Service eingebaut, um zu verifizieren, dass die Anfragen tatsächlich auf die Instanzen verteilt werden.

Für das Frontend habe ich eine Vue.js Webapp zum größten Teil von KI generieren lassen, was sehr gut funktioniert hat. Ich musste nur gelegentlich noch Anpassungen vornehmen, um die Kompatibilität mit den Service-Endpunkten zu gewährleisten. Der Vorteil einer Vue.js Web-App (Ebenso React oder Angular) ist, dass der Webserver nur beim erstmaligen Zugriff angefragt wird, und danach alles lokal beim Nutzer abläuft. Daher wird die Last auf dem Webserver minimiert und er muss nicht notwendigerweise horizontal skaliert werden. Es wäre aber problemlos möglich, auch hier einen Load-Balancer dazwischen zu schalten und beliebig viele Webserver auf der ganzen Welt zu betreiben, die die App bereitstellen. Ich habe mir das in diesem Fall gespart, da mein Rechner bereits mit den nun vorhandenen acht Docker-Containern ziemlich zu kämpfen hat. Die größte Systemlast wird dabei aber durch die drei Cassandra-Instanzen verursacht.

Verwendung der Docker-Compose Anwendung

Die Compose-Anwendung öffnet zwei Ports auf der Host-Maschine. Über **Port 8080** ist der Webserver zu erreichen, auf dem die Vue-Applikation liegt. Auf **Port 3000** erreicht man den Load Balancer, der die Anfragen entsprechend auf die Wishlist-Service-Instanzen aufteilt.

API-Dokumentation

- `GET /api/ping`
  Löst eine Log-Nachricht im Webservice aus. So kann man in Docker-Compose zum Beispiel sehen, welche Service-Instanz den Call erhalten hat.
- `GET /api/getWishlists`
  Gibt eine Liste aller Wunschlisten mit ID und Name zurück.
  Returns:

  ```
  {
          "wishlist_id": uuid,
          "name": string
  }[]
  ```

- `POST /api/submitWishlist`
  Trägt eine Wunschliste inklusive Wünschen in die Datenbank ein.
  Accepts:

  ```
  {
          "name": string,
          "wishes" string[]
  }
  ```

- `POST /api/beginProcessing`
  Gibt einen Wunsch zurück, der aktuell auf *Formuliert* steht, und setzt ihn auf *In Bearbeitung*.
- `PATCH /api/loadOnSleigh/<wish_id>`
  Setzt den gegebenen Wunsch auf *In Zustellung*.
- `GET /api/getWishlistByName?name=<name>`
  Gibt die Wunschliste mit allen Wünschen für eine Person zurück.
- `PATCH /api/setWishDelivered/<wish_id>`
  Setzt den gegebenen Wunsch auf *Zugestellt*.

Performance-Messung

Da mein Rechner allein durch die Ausführung aller Komponenten von XMasWishes sehr überfordert ist, ist dieses Beispiel vermutlich sehr verzerrt. Mein Rechner alleine ist definitiv nicht dazu in der Lage, so viele Anfragen pro Sekunde zu stellen wie es mehrere Milliarden Menschen könnten. Trotzdem habe ich einen kleinen Versuchsaufbau eingerichtet: Ich lasse in einem Java-Programm je Versuchsdurchlauf 15 Sekunden lang immer wieder den `getWishlists`-Endpoint aufrufen, und zähle die Anzahl der Aufrufe. Das ganze führe ich 40 mal durch und berechne im Anschluss ein paar Statistiken dazu.

Ergebnisse

Folgende Abbildungen zeigen die Ergebnisse meiner Auswertung. Aufgrund der vielen Durchläufe konnte ich die Requests pro 15 Sekunden recht gut einschätzen. Im Histogramm ist das 95%-Konfidenzintervall in Blau eingezeichnet.

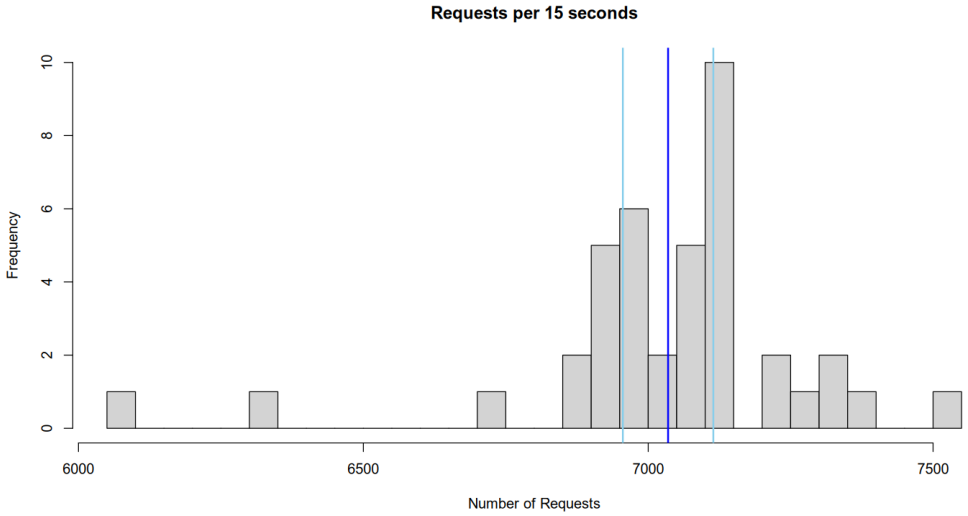

Abbildung 2: Histogramm der Ergebnisse

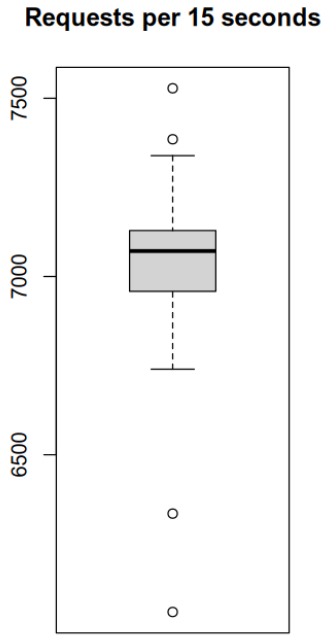

Abbildung 3: Boxplot der Ergebnisse

```
> summary(results)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   6059    6966    7072    7035    7128    7528
```

Der Durchschnittswert liegt bei etwa 7000 Requests pro 15 Sekunden. Pro Sekunde wären das also etwa 466 Anfragen pro Sekunde. Die errechnete Last in Phase 3 (Auslieferung) beträgt

allerdings 120000 Anfragen pro Sekunde. Da die hier verwendeten Technologien linear horizontal skalieren, bräuchte man $120000 \div 466 \approx 258$ mal so viele Ressourcen, wie ich auf meinem Laptop zur Verfügung habe. Der Vollständigkeit halber hier nochmal meine Systemspezifikationen:

Merkmal	**Spezifikation**
CPU-Bezeichnung	AMD Ryzen 5 4500U with Radeon Graphics
CPU-Architektur	x86 (AMD Renoir)
CPU-Fertigungsverfahren	7 nm (TSMC)
Anzahl Kerne / Threads	6 / 6
Basistaktfrequenz	2.3 GHz
Maximale Boost-Taktfrequenz	4.0 GHz
L1-Cache	384 KB
L2-Cache	3 MB
L3-Cache	8 MB
TDP (Thermal Design Power)	15 Watt
Maximale Temperatur	105 °C
Arbeitsspeicher	2x4GB DDR4-3200 (Dual Channel)
Erscheinungsdatum	07.01.2020