FACHHOCHSCHULE
ERFURT UNIVERSITY
OF APPLIED SCIENCES
Angewandte
Informatik

Parallelisierung und Zugriffssynchronisation bei voneinander abhängigen Ressourcen-Gittern

Bernd Schmidt

5. Juli 2015

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Aufgabenstellung

Im Rahmen des Projekts Ascendancy in Mobile Computing wurde ein Spiel entwickelt, bei dem tausende von Benutzern zusammen beziehungsweise gegeneinander spielen sollen. Dies erfordert eine Server-Anwendung, welche in C# implementiert wurde und stark parallelisiert werden muss.

Die dazugehörige Client-Anwendung sendet Änderungswünsche an den Server. Der Server bearbeitet diese Änderungswünsche und ändert seinen Datenbestand entsprechend. Gegebenenfalls wird die gewünschte Änderung noch angepasst oder verworfen, was für die Aufgabenstellung aber irrelevant ist.

Der Client muss stets einen möglichst aktuellen Stand der Daten erhalten können. Ebenso soll es für den Client möglich sein, sämtliche Änderungen seit einem vorherigen Datenbestand zu erhalten, sofern dieser nicht länger zurück liegt.

Der Datenbestand in der aktuellen Version beträgt etwa 1 Milliarde Datensätze. Diese Datensätze sind je nach Änderungswunsch voneinander abhängig. Wird ein Datensatz geändert, müssen gegebenenfalls mehrere andere Datensätze überprüft und teilweise auch geändert werden.

Notwendige Kriterien für die Parallelisierung der Server-Anwendung sind daher:

- Hohe Anzahl von Client-Lesezugriffen
- Viele parallele Schreibzugriffe
- Konsistenz des an den Client gesendeten Datenbestands und Änderungen
- Strikte Vermeidung von Verklemmungen
- Parallele Abarbeitung der Client-Änderungswünsche

Die Kriterien werden im Abschnitt 4.2 genauer analysiert und spezifiziert.

2 Ascendancy

Ascendancy soll ein Spiel für Smartphones werden. Es orientiert sich dabei an Spielen wie Battle of Wesnoth, Civilisation, Endless Legend und Heroes of Might and Magic.

Im Gegensatz zu den genannten Spielen, soll man Ascendancy in und auf der realen Welt spielen. Dazu wurden Geodaten von openstreetmap.org in eine Spielwelt konvertiert. Diese Spielwelt besteht aus vielen Feldern (Datensatz), auf denen Einheiten und Gebäude stehen sollen. Einheiten können sich bewegen (Aktion). Falls sich eine Einheit bewegt, müssen die Felder in der Nähe überprüft und das Start- und Endfeld entsprechend leer beziehungsweise mit der jeweiligen Einheit gesetzt werden.

Ein Feld besitzt eine Position in der Spielwelt. Auf diesem Feld kann eine Einheit stehen. Eine Aktion, zum Beispiel das Bewegen einer Einheit, besteht aus der Information welche Einheit wohin bewegt werden soll.

3 Grundlagen

3.1 Entwicklungsumgebung / Produktivumgebung

Als verwendete Programmiersprache kam C# zum Einsatz. Die Entwicklung selbst fand unter folgendem System statt:

- MonoDevelop 5.9.4
- Mono JIT compiler version 4.0.1
- Mono.WebServer2.dll 0.4.0.0
- Mint 17 Qiana
- Linux 3.13.0-24-generic 47-Ubuntu x86_64
- KDE: 4.13.3

Auf dem Server befanden sich folgende Konfigurationen:

- Mono JIT compiler version 4.0.1
- Mono.WebServer2.dll 0.4.0.0
- Debian Jessie
- Linux 3.2.0-4-amd64 Debian x86_64

3.2 Synchronisationsmechanismen

Die System.Threading Bibliothek von C# bietet die üblichen Mechanismen für Threading an. [MSD52]

- Thread

  Threads sind notwendig, um die genannte Parallelisierung zu erreichen. Da ein ASP.NET MVC Server verwendet wird, ist automatisch jeder Zugriff auf diesem ein eigener Thread.

- Interlocked

  Interlocked bietet atomare Operationen wie das Addieren zweier Zahlen oder Inkrementieren an. Im Gegensatz zu den herkömmlichen Operatoren sind diese garantiert Thread-Safe.

- Monitor

  Steuert den Zugriff auf Objekte indem dem Thread eine Sperre für das entsprechende Objekt zugeteilt wird. Kann wartende Threads über Änderungen des gesperrten Objekts informieren.

- Mutex

  Ein Synchronisationsmechanismus, der sicherstellt, dass eine Ressource nur von einem Thread genutzt wird.

- ReaderWriterLock

  Stellt einen Synchronisationsmechanismus bereit, der mehrere lesende und einen schreibenden Zugriff erlaubt.

- Semaphore

  Ähnlich dem Mutex stellt dieser Synchronisationsmechanismus sicher, dass eine Ressource nur von einer maximalen Anzahl von Threads genutzt werden kann.

- Timer

  Führt eine Methode nach einem gegebenen Intervall aus, gegebenenfalls mehrfach.

- Diverse Exceptions

  Diese beinhalten für die Lösung des Problems keine relevanten Methoden oder Mechanismen.

Unter anderem stehen noch Transaktionen und Locks zur Verfügung.

3.3 Feld

Ein Feld ist eine kleine Fläche der realen Welt, wie in Abbildung 3.1 zu sehen ist. Die Größe variiert je nach Längen- und Breitengrad, beträgt aber in Deutschland etwa 3 * 3 Meter. In Ascendancy wird es intern als Zweidimensionale Position gehandhabt. Es können auf einem Feld Einheiten stehen, dies ist gegenwärtig die einzige Information die sich ändern kann.

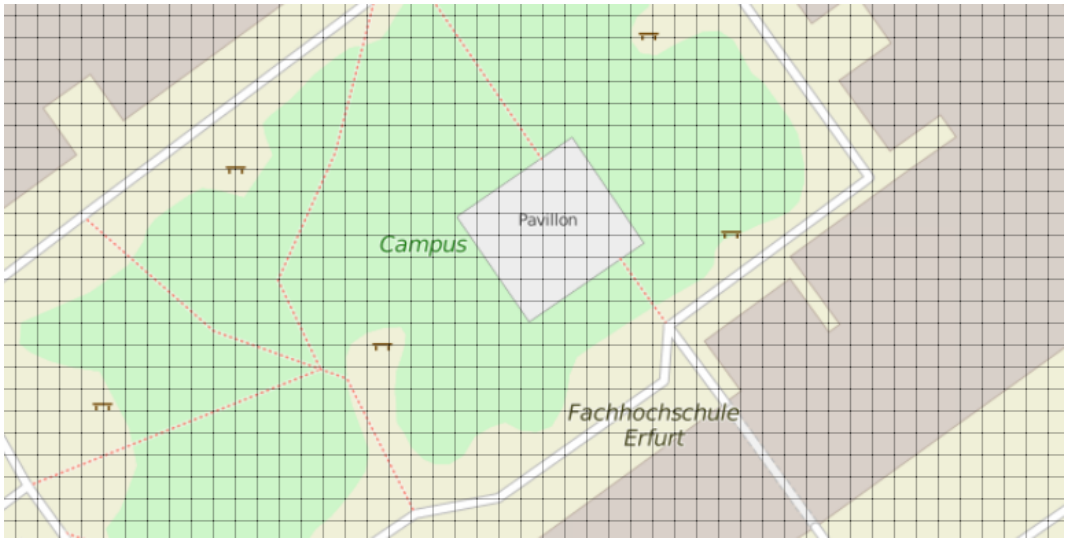

Abbildung 3.1: Die Welt besteht aus verschiedenen Feldern. (Bild editiert, von [OSM57])

Da es mehrere Milliarden Felder gibt, werden nur jene geladen, welche auch benutzt werden.

Es existieren keine Lücken zwischen Feldern. Beim Zugriff auf nicht geladene oder nicht-existente Felder wird eines vom Typ Ungültig erzeugt. Somit ist jedes Feld definiert.

3.4 Aktion

Aktionen sind Änderungswünsche des Clients an den Datenbestand des Servers. Der Server arbeitet diese ab und informiert die Clients über die stattgefundenen Aktionen.

Als einfaches Beispiel sei hier das Bewegen einer Einheit genannt. Der Client schickt einen Befehl an den Server, dass dieser eine Einheit bewegen möchte. Der Server prüft ob, es möglich ist und berechnet dann den Pfad. Der Client erhält bei einer späteren Anfrage die Information, ob er die Einheit auch bewegen darf beziehungsweise wie weit.

Beim Berechnen des Pfades darf sich der für die Aktion relevante Datenbestand nicht ändern. Gegebenenfalls muss die Aktion allerdings selbst Änderungen vornehmen, um die Einheit umzusetzen.

Welche Felder von einer Aktion betroffen sind, ist von der jeweiligen Aktion abhängig. Das Bewegen einer Einheit findet im Rahmen von etwa +- 32 Feldern statt und kann somit bis zu 4 der in 4.1.1 beschriebenen Regionen betreffen, wenn eine Einheit U-förmig an der Grenze der vier Regionen läuft.

4 Konzept

4.1 Vereinfachungen

4.1.1 Regionales Bündeln von Datensätzen

Jedes Feld hat eine Position. Es können räumlich mehrere Felder zu einem Datenverbund, genannt Region, zusammengefasst werden.

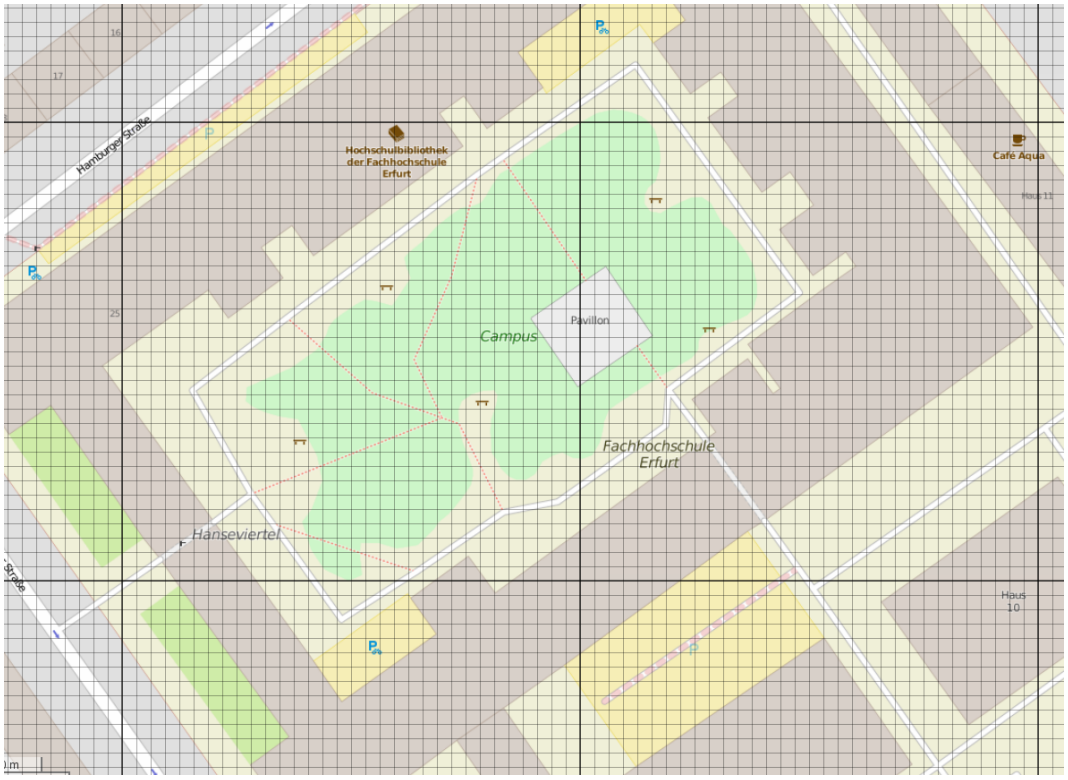

Abbildung 4.1: 32 x 32 Felder werden zu einer Region zusammengefasst

Wenn 32 * 32 Felder wie in Abbildung 4.1 zusammengefasst werden, sind es 1024 pro Region. Das Grundproblem bleibt das Gleiche, statt dem Zustand des Feldes ist im Folgenden der Zustand der Region relevant.

Beim Bewegen einer Einheit ist vorher nicht klar, wie viele und welche Felder relevant sein können. Falls die Einheit sich maximal +-16 Felder bewegen darf, müssen etwa 33 * 33 = 1089 Datensätze betrachtet werden. Nutzt man Regionen, wären dies maximal 4 Datenverbände, wenn die Einheit sich U-förmig an den Grenzen der Regionen bewegen würde.

Dies wird zum Beispiel in Abschnitt 5.1.2 relevant, wo es einen deutlichen Unterschied macht, ob vier oder 1089 Mutexe gesperrt werden müssen.

Dies ermöglicht eine deutliche Reduzierung und bessere Handhabung der zu betrachtenden Daten.

4.1.2 Aktion-Warteschlange

Für jede Anfrage öffnet ein Server einen eigenen Thread. Wenn zu viele Threads geöffnet sind, werden weitere Anfragen verworfen.

Da das Ausführen einer Aktion vergleichbar lange dauert, könnten zu viele Aktion-Anfragen den Server blockieren. Um dieses Problem zu umgehen und um mehr Kontrolle über die abgearbeiteten Aktionen zu erhalten, wird eine Warteschlange benutzt.

Jede auszuführende Aktion wird in eine FIFO-Warteschlange (First In - First Out) eingetragen. Eine definierte Anzahl so genannter Worker-Threads arbeitet die Aktions-Warteschlange ab. Falls keine Elemente in der Warteschlange sind, so warten diese Threads, bis neue eingetragen werden.

Der Client wird durch diese Umsetzung allerdings nicht mehr umgehend informiert, ob und inwieweit die Aktion ausgeführt wurde. Dies wird der Client in einer weiteren Anfrage erfahren.

4.2 Spezifikation der notwendigen Kriterien

Wie bereits in der Aufgabenstellung in Abschnitt 1 erläutert, müssen die Kriterien weiter spezifiziert werden. Die strikte Vermeidung von Verklemmungen ist notwendig, ebenso muss stets ein konsistenter Datenbestand existieren. Allerdings muss die maximale Anzahl von Lese- und Schreibzugriffen festgelegt werden.

Falls Werte geschätzt werden, wird stets von einem möglichst hohen Wert ausgegangen.

Eine Region besteht aktuell aus 32 * 32 Felder = 1024 Felder.

Die Regeln aus Ascendancy legen fest, dass ein Feld in Deutschland etwas mehr als 3m * 3m beträgt. Dies ist vom Breiten- und Längengrad abhängig. Eine Region ist demnach etwa 100m * 100m groß. Ein Spieler läd stets die umliegenden 25 Regionen, dies sind circa 500m * 500m = 0.250 km².

Falls jedes der 1024 Felder in einer Region belegt ist, können pro Runde, pro Region maximal 1024 verschiedene Aktionen existieren, die zu verarbeiten sind.

Aus den verschiedenen Werten ergeben sich folgende Parameter:

Dieses Test-Szenario simuliert die maximale Last in einem kleinen eingeschränkten Gelände.

Test a)

- In jeweils 25 Regionen
- Ausführung von 1024 Aktionen pro Region
- Jede Aktion soll eine maximale Anzahl von Regionen (4) betreffen (Abschnitt 3.4)

Für einen aussagekräftigen Belastungstest wird von einem größeren Gelände ausgegangen.

Test b)

- 400 Regionen
- Ausführung von 1024 Aktionen pro Region
- Jede Aktion soll eine maximale Anzahl von Regionen (4) betreffen (Abschnitt 3.4)

Um realistische Umgebungen zu simulieren, wird die Anzahl der Regionen deutlich erhöht. Die auszuführenden Aktionen werden reduziert, da nicht immer in jeder Region etwas stattfinden muss.

Test c)

- In jeweils 10000 Regionen
- Ausführung von 32 Aktionen pro Region
- Jede Aktion soll eine maximale Anzahl von Regionen (4) betreffen (Abschnitt 3.4)

Zum Schluss soll getestet werden, wie die Anzahl der zu reservierenden Ressourcen sich auf die Zeit auswirkt. Dies ist ein Spezialfall, der seltener Auftreten sollte.

Test d)

- In jeweils 25 Regionen
- Ausführung von 1024 Aktionen pro Region
- Jede Aktion soll nur eine Region betreffen

Bei den Implementationen in Ascendancy wurden stets zwei Worker-Threads gestartet.

Für die Anfragen wurde ein C#-Programm geschrieben, welches die entsprechende Anzahl an Anfragen erzeugt und die zur Abarbeitung benötigte Zeit misst. Nach jeder Anfrage wird der geänderte Teil zudem gelesen.

Einflussfaktoren wie ein Zugriff auf die Festplatte wurden eliminiert, indem die Regionen vor der Zeitmessung einmal vorgeladen wurden.

4.3 Grafische Veranschaulichkeit des Problems

Die vorliegenden Abbildungen 4.2 - 4.3 zeigen grob den zeitlichen Ablauf.

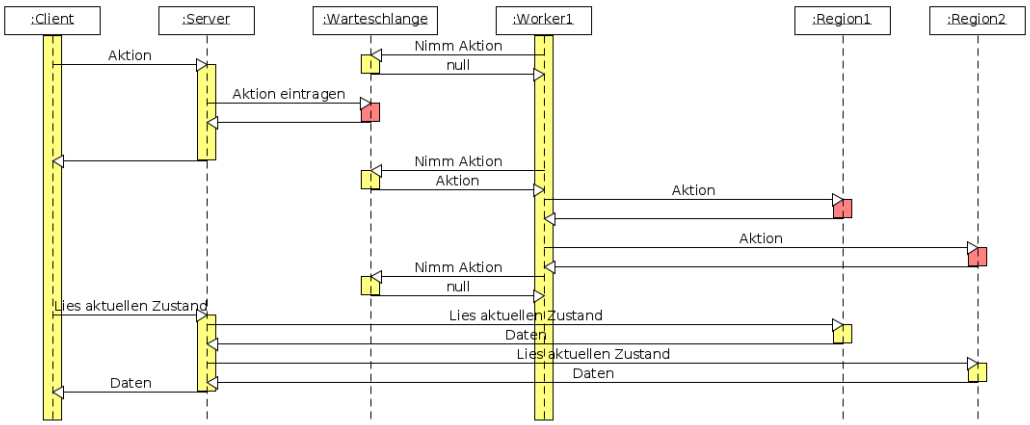

Abbildung 4.2: Ideale Situation

Der Client schickt eine Aktion und versucht im folgenden den aktuellen Zustand der Regionen zu lesen. Falls dies im falschen zeitlichen Ablauf geschieht, kann es zu inkonsistenten Datenständen auf dem Client kommen. Dies soll verhindert werden.

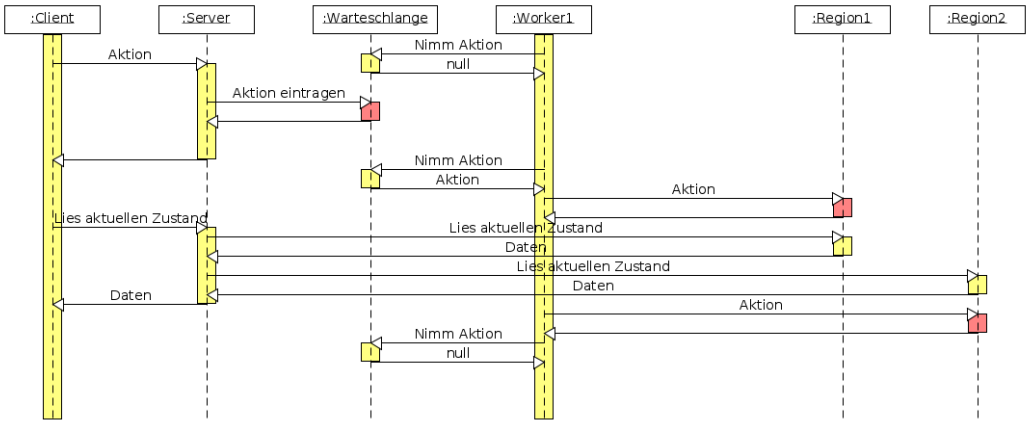

Abbildung 4.3: Client erhält inkonsistente Daten

In der Abbildung 4.3 würde der Client den neuen Zustand einer Region erhalten, jedoch den alten einer anderen Region. Am einfachsten kann man sich das wie folgt vorstellen:

- Der Client möchte eine Einheit bewegen, von Region1 nach Region2
- Der Worker entfernt die Einheit von Region1
- Der Client liest Region1 und Region2, in keinem der beiden befindet sich die Einheit
- Der Worker setzt die Einheit auf Region2

Der Client besitzt nun einen inkonsistenten Datenstand. Dasselbe Problem tritt auch bei mehreren Workern und mehreren Clients auf.

5 Umsetzung

5.1 Schreiber-Leser-Mechanismus

5.1.1 Globaler Mutex

Als einfachste, simpelste Lösung bietet es sich an, einen einzigen Mutex zu verwenden. Jeder Schreibzugriff sperrt sämtliche Felder.

Damit die Leser allerdings einen konsistenten Datenbestand erhalten, dürfen diese nur lesen, wenn gerade keine Aktion ausgeführt wird. Durch Nutzung eines WriterReaderLock, statt eines Mutex, kann dies mit mehreren Lesern realisiert werden.

Dieser Ansatz ist garantiert frei von Verklemmungen und einfach umsetzbar. Zudem erhalten die Leser konsistente Daten, weshalb dieser Ansatz als wichtiger Vergleichswert dient.

Wie in Abbildung 5.1 sichtbar, werden bei jedem Zugriff sämtliche Daten gesperrt. Es kann stets nur eine einzige Aktion zur gleichen Zeit ausgeführt werden. Bei einer derartigen Umsetzung werden allerdings nicht mehr mehrere Worker-Threads benötigt, es würde ein einziger reichen.

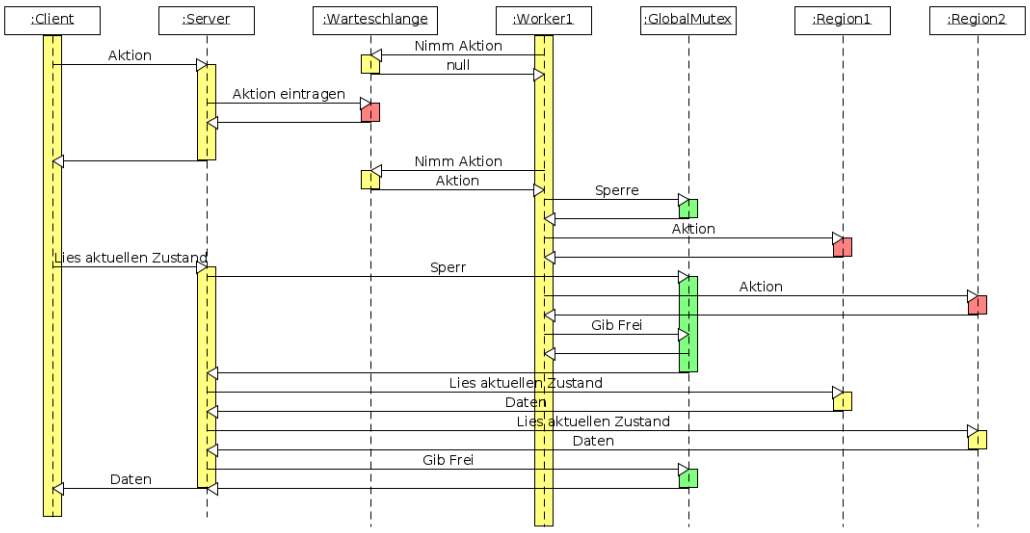

Abbildung 5.1: Durch einen globalen Mutex darf immer nur ein Thread Lesen oder Schreiben

Bei der Umsetzung des globalen Mutex ergeben sich folgende Zeiten:

Test a) 650 Millisekunden

Test b) 13500 Millisekunden

Test c) 11300 Millisekunden

Test d) 420 Millisekunden

5.1.2 Regionaler Mutex

Um ein feineres Sperren als einen globalen Mutex zu gewährleisten, könnte jede Region einen Mutex erhalten. Vor einem Schreibzugriff werden alle benötigten Regionen komplett gesperrt. Danach werden die Mutexe wieder freigegeben.

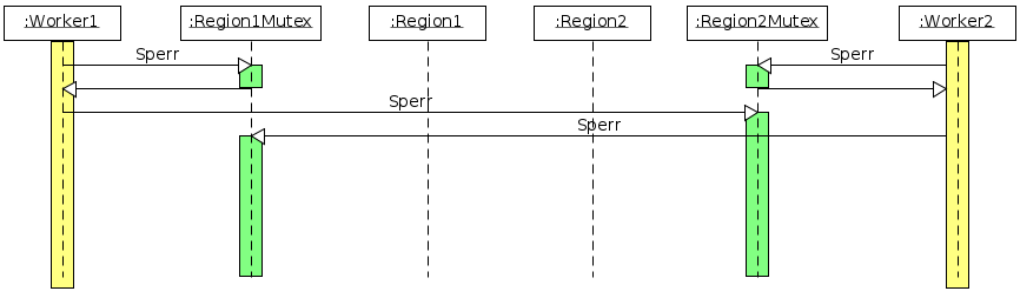

Abbildung 5.2: Regionale Mutexe können zu Verklemmungen führen

Wie in Abbildung 5.2 zu sehen, kann dieser Fall zu Verklemmungen führen, falls zwei Threads dieselben Regionen sperren möchten.

Für dieses Problem existieren zwei Lösungen:

- Deterministische Region Reservierung

  Bei jeder Aktion kann eine Aussage darüber getroffen, welche Regionen benötigt werden. Falls diese Regionen stets in einer deterministischen Reihenfolge reserviert werden, könnte keine Verklemmung entstehen.

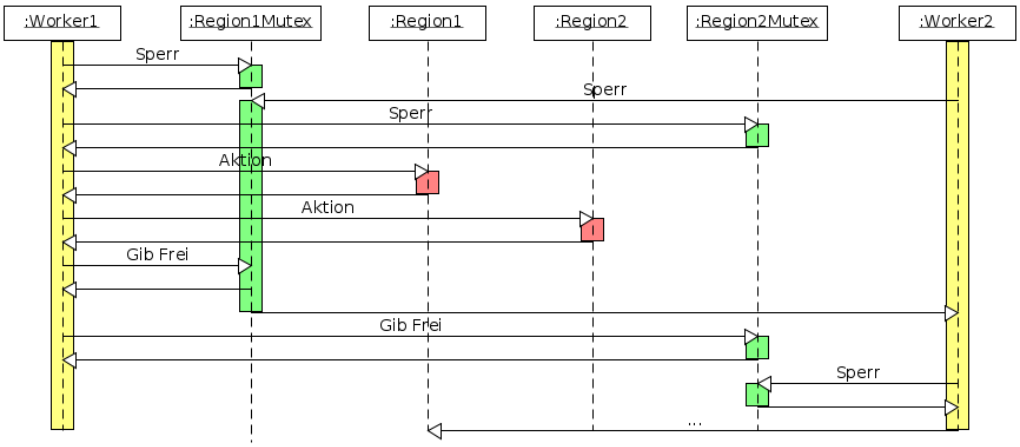

Abbildung 5.3: Es wird auf den ersten Worker gewartet

- No Wait

  Ein Worker-Thread versucht, die von der Aktion benötigten Ressourcen zu sperren. Falls eine Ressource nicht gesperrt werden kann, weil ein anderer Worker-Thread diese gerade benutzt, so wird die entsprechende Aktion abgebrochen und an das Ende der Warteschlange gesetzt. Es können dann andere Aktionen abgearbeitet werden.

Je höher die Anzahl der benötigten Ressourcen, desto größer ist auch die Wahrscheinlichkeit, dass eine Aktion wieder in der Warteschlange endet. Zwei Aktionen, welche dieselben Ressourcen benötigen, aber stets zur gleichen Zeit unterschiedliche Ressourcen reservieren, könnten in einen Livelock enden.

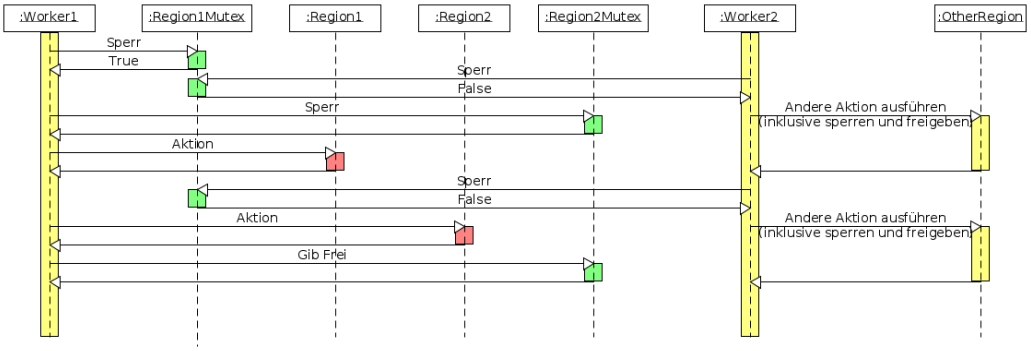

Abbildung 5.4: Wenn ein Worker nicht sperren kann, führt er eine andere Aktion aus

Keine der beiden Lösungen ist optimal. Es wurden jedoch beide implementiert und gemessen:

Test a) 5000 Millisekunden

Test b) 17830 Millisekunden

Test c) 8700 Millisekunden

Test d) 240 Millisekunden

Es wurde im speziellen Fall ein ReaderWriterLock implementiert, damit die Leser nur einen konsistenten Stand lesen können. Die Leser sperren die Regionen relativ lange, da sie sich anhand der Daten eine neue Liste zusammen bauen. Die Daten dürfen sich nicht ändern, weshalb es in den genannten Synchronisationsmechanismen geschehen muss. Diese könnten zwar auch kopiert werden, um außerhalb vom ReaderWriterLock mit ihnen zu arbeiten, dies hat aber noch länger gedauert.

Da die Sperrung relativ lange dauert, ist es Wahrscheinlich, dass es zu längeren Blockierungen führt. Dies bestätigt auch die Geschwindigkeit von Test c) und Test d), bei denen es kaum noch zu Blockierungen kommt.

5.2 Vorhalten älterer Welt-Stände

Eine weitere Möglichkeit, um dem Client stets konsistente Daten zu liefern, besteht darin, einen älteren konsistenten Datenbestand bereitzustellen. Dieser wird im folgenden Read-World genannt.

Sämtliche Änderungen des Datenbestands finden in der Write-World statt. In dieser werden die in 5.1.2 genannten regionalen Sperrungen verwendet. Dabei kommen beide Lösungen, sowohl die No-Wait wie auch die deterministische Reihenfolge zur Reservierung gleichzeitig zum Einsatz.

Der große Vorteil dieser Technik ist, dass die Leser von den Schreibern getrennt sind. Beide agieren erst einmal unabhängig voneinander.

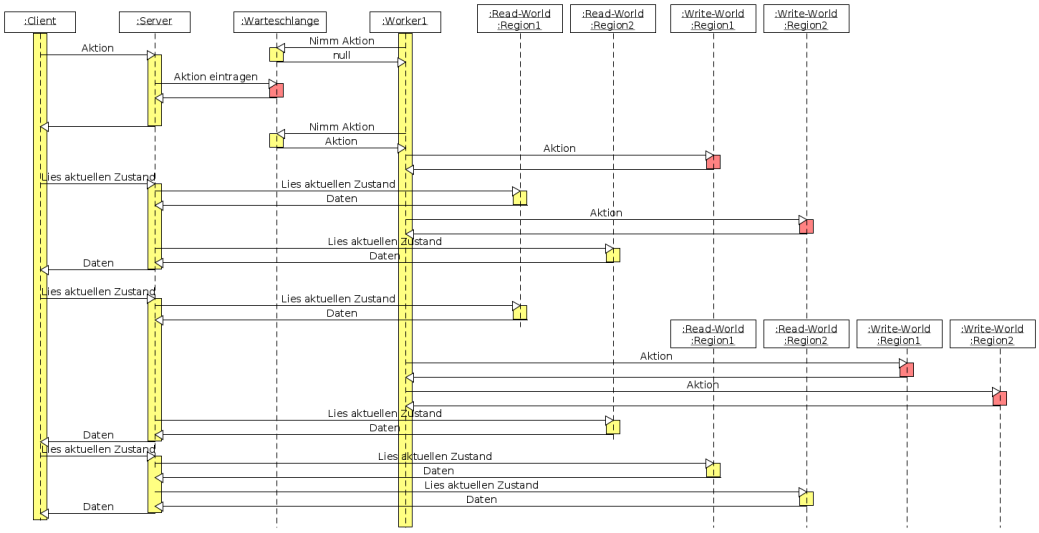

Abbildung 5.5: Der Client greift auf einen alten Stand zu

Der in Abbildung 5.5 gezeigte Client greift auf einen Datenbestand zu. Wenn während des Zugriffs der Datenbestand wechselt, würde der Client weiterhin auf den selben Datenbestand zugreifen können, da eine lokale Referenz auf dem alten Datenbestand gehalten werden kann.

Einzelne Regionen können nicht ausgetauscht werden, da ohne entsprechende Sperrung inkonsistente Datenbestände die Folge sein könnten.

Der Austausch der kompletten Read-World durch eine aktuelle Write-World ist allerdings ohne Sperrung denkbar. Es ist möglich, lediglich die Referenz des Read-World zu ändern. Alle Threads würden bei weiteren Zugriffen nur noch auf die neue Read-World zugreifen.

Bevor allerdings die Referenz geändert werden kann, muss eine neue Write-World erstellt werden.

Dafür bestehen verschiedene Ansätze:

- Kopieren der bestehenden Write-World inklusive sämtlicher Regionen. Dadurch entsteht eine neue Welt, auf der normal gearbeitet werden kann. Diese Lösung kann bei größeren Datenmengen zu lange dauern.
- Es könnte nur die erste Ebene des Write-World, ohne die Ressourcen, kopiert werden. Da die Referenzen auf die alten Ressourcen erhalten bleiben, würde sich bei einem Schreibzugriff der Zustand in der Read-World ändern. Vor einem Schreibzugriff muss eine Referenz kopiert und neu in das Write-World eingetragen werden.
- Da das Anlegen einer flachen Kopie der Write-World ebenfalls lange dauern könnte, besteht die Möglichkeit, diese leer zu initialisieren. Falls auf eine Region zugegriffen wird, wird diese aus dem Read-World kopiert. Der aktuellsten Write-World fehlen dabei Regionen, jene, auf die noch nicht zugegriffen wurde. Es wird eine Speicherung dieser Regionen nötig, da diese beim Austausch der Read-World sonst abhanden kämen.

Die letzte Lösung wird aus Performance-Gründen bevorzugt. Für den späteren Verlauf der Anwendung wird ohnehin ein Algorithmus von Nöten sein, der nicht mehr benötigte Regionen findet und diese sichert.

5.2.1 Problemstellung Schreiber

Falls eine Aktion etwas an der Write-World ändert, diese aber zum gleichen Zeitpunkt als Read-World gesetzt wird, würde die Aktion etwas an einem Objekt ändern, welches sich nicht ändern sollte.

Dies wäre theoretisch mit Hilfe einer einfachen Umschalt-Sperre vermeidbar. Dann müssten die Worker angehalten werden, bevor die Read-World ausgetauscht wird. Da dies eine kurzzeitige Unterbrechung der Arbeit ist, wird eine andere Lösung gesucht.

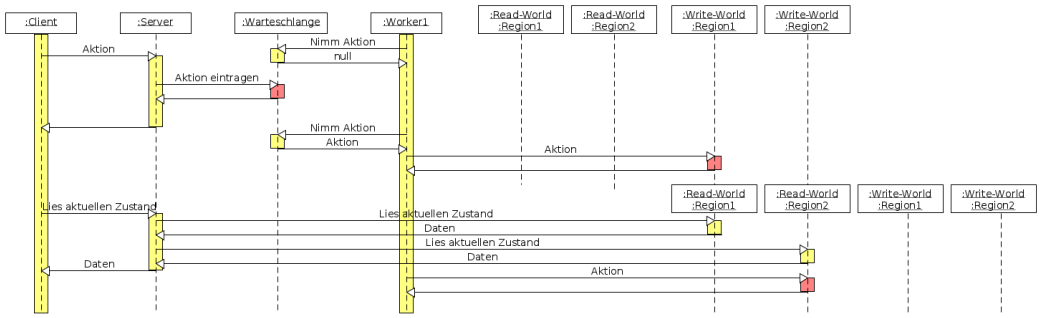

Abbildung 5.6: Der Worker arbeitet auf einem Stand, der nur lesbar sein sollte

Es muss gewartet werden, bis sämtliche Aktionen abgearbeitet wurden. Daher wird vor dem Wechsel der Welt bereits eine neue Welt erstellt, auf der weitere Aktionen ausgeführt werden können. Während die Applikation auf das Beenden der älteren Aktionen wartet, können so bereits neue ausgeführt werden.

Wie in Abschnitt 5.2 erläutert, sollte die Welt leer sein und sich erst mit weiteren Anfragen füllen. Dabei wird auf die alte Write-World zurückgegriffen und entsprechende Regionen kopiert. Falls der Mutex einer Region gesperrt ist, wird diese nicht kopiert. Damit wird verhindert, dass die neue Write-World entsprechende Regionen läd, auf denen noch alte Aktionen ausgeführt werden.

Die Reihenfolge:

- Neue Write-World wird erstellt
- Neue Aktionen verwenden die neue Write-World
- Es wird gewartet, bis sämtliche Aktionen auf der alten Write-World abgeschlossen sind
- Die alte Write-World wird als Read-World gesetzt
- Die neue Write-World wird als Write-World gesetzt

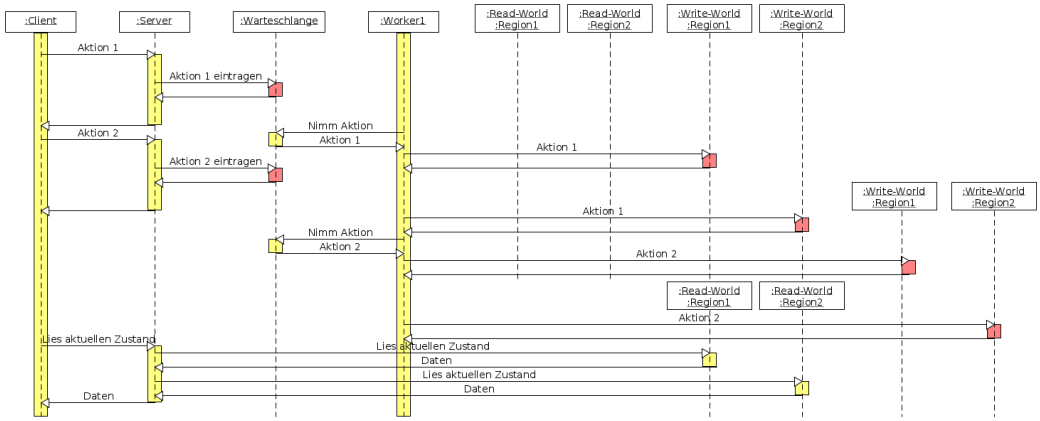

Abbildung 5.7: Aktionen werden stets in der neusten Write-World ausgeführt

5.2.2 Problemstellung Leser

Der Leser erhält einen älteren Stand der Anwendung. Dieser Stand kann, je nach Einstellung, wenige Sekunden oder Minuten alt sein.

Um zu alte Stände zu vermeiden, exisitieren zwei Möglichkeiten.

- Aktualisieren der Read-World jede Sekunde

  Dies dürfte bei zehntausenden geladenen Regionen zu langsam sein oder zu viel Arbeitsspeicher benötigen. Jede Region müsste entweder kopiert, oder alte Zustände im Speicher gehalten werden.

- Aktionsliste in der Read-World

  Ändert man die Write-World, speichert man die Aktion jedoch in einer Liste der Read-World, kann der Client sich den aktuellen Status der Welt selbst berechnen. Dabei tritt das Problem, welches bei den Tests von 5.1.2 beschrieben wurde, wieder auf. Wenn die Änderungen in der Read-World gespeichert werden, muss es für das Schreiben und Lesen wieder eine Synchronisation geben, bei dem der Lesezugriff blockierend wirkt.

Diese Umsetzung bietet für Ascendancy leider keinen Nutzen. Falls eine Anwendung parallelisiert werden soll, bei der es nicht auf Sekunden oder Minuten ankommt, so ist dieser Ansatz empfehlenswert.

5.3 Vermeidung von Blockierungen

Es ist deutlich, dass mehrere Threads sich gegenseitig blockieren. Es muss daher ein Mechanismus gefunden werden, welcher es den Threads ermöglicht möglichst blockierungsfrei zu arbeiten. Eine erste Idee bestand darin, je eine Menge von Regionen zusammen zu fassen und diesen eine eigene Warteschlange und einen eigenen Thread zu geben. Dies ist zwar prinzipiell möglich, aber aufgrund der Menge der benötigten Threads unpraktikabel.

Der folgende Algorithmus wurde selbst entwickelt, um Blockierungen zu vermeiden. Dabei wurde überlegt, wie ein Thread eine bestimmte Fläche abarbeiten kann, welche sich je nach Auslastungslage ändern könnte.

Wie in 5.1.2 erläutert, wurden hier bei jeder Region ein ReaderWriterLock verwendet.

- Es existieren n Threads
- Es existieren n Warteschlangen
- Jeder Thread arbeitet eine Warteschlange ab
- Warteschlangen besitzen eine Position
- Die Position der Warteschlange errechnet sich aus dem Mittelwert der enthaltenen Aktionen
- Die Position einer Aktion ist, wo der erste relevante Zugriff erfolgt
- Eine Aktion wird in die erste leere Warteschlange eingefügt
- Beim Einhängen/Aushängen in eine Warteschlange wird eine neue Position berechnet
- Existiert keine leere Warteschlange, wird errechnet, welche Warteschlange am nächsten zu der Position der Aktion ist

Da es sich um eine zweidimensionale Position handelt, wird der Mittelwert jeweils über x und y berechnet. Um den Mittelwert der Warteschlangen zu erhalten, welcher am nächsten ist, wird der Satz des Pythagoras verwendet.

So wird gewährleistet, dass sich alle Threads um alle Aktionen kümmern. Leerlaufende Threads werden vermieden, da Aktionen immer zuerst in leere Warteschlangen eingefügt werden.

Dabei kommen Threads ohne Arbeit mit hoher Wahrscheinlichkeit an Stellen, an denen eine hohe Last herrscht. Dies sollte eine möglichst optimale Lastaufteilung ermöglichen. Blockierungen werden allerdings vermieden, weil diese sich möglichst maximal voneinander entfernen.

Hierbei muss berücksichtigt werden, dass der Zugriff auf den Mittelwert und Berechnungen ebenfalls von mehreren Clients gleichzeitig erfolgen könnte. Dieser Zugriff muss ebenfalls Synchronisiert werden, dies wird mit einem Mutex realisiert. Da jeder Thread maximal eine Warteschlange reserviert kann es zu keiner Verklemmung kommen.

Es wurde als Warteschlange bislang ein ConcurrentQueue verwendet. Durch die Verwendung des Mutex konnte diese durch eine einfache Queue ausgetauscht werden. Die Benutzung des Mutex hat keinerlei negative zeitliche Auswirkungen, da die ConcurrentQueue auch Synchronisationsmechanismen verwendete [MSD00].

Test a) 460 Millisekunden

Test b) 9400 Millisekunden

Test c) 7800 Millisekunden

Test d) 220 Millisekunden

Diese Lösung ist optimal.

6 Zusammenfassung

6.1 Fazit

Es war zweifelhaft, ob Synchronisationsmechanismen gefunden werden können, welche die Voraussetzungen für das System erfüllen.

Die Lösung zur Vermeidung von Blockierungen in Abschnitt 5.3 ist in jeder betrachtenden Situation die beste Wahl. Eine Integration in den Hauptentwicklungszweig von Ascendancy war erfolgreich.

Ob die Performance wirklich ausreicht wird sich leider erst am Ende des Moduls Mobile Computing 2 zeigen, da erst dann die ersten größeren Tests und eine erste Veröffentlichung stattfinden sollen.

Literaturverzeichnis

[MSD52] Microsoft Developer Network: *System.Threading Namespace*. Version: 01.07.2015 13:52. https://msdn.microsoft.com/de-de/library/system.threading%28v=vs.110%29.aspx

[MSD00] Microsoft Developer Network: *ConcurrentQueue Quelltext*. Version: 01.07.2015 14:00. http://referencesource.microsoft.com/#mscorlib/system/Collections/Concurrent/ConcurrentQueue.cs.html

[OSM57] OpenStreetMap Contributors: *OpenStreetMap Erfurt*. Version: 20.06.2015 20:57. http://www.openstreetmap.org/#map=19/50.98512/11.04260