Seminarausarbeitung AW1

Kjell Otto

Clojure
concurrency revisited

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

Heutige CPUs unterscheiden sich hauptsächlich durch die Anzahl an Kernen innerhalb der CPU und nicht mehr, wie vor einigen Jahren noch, durch die Taktrate. Das Problem mit der Taktsteigerung ist auf physikalische Gesetze zurückzuführen, die bei einer kleinen Erhöhung der Taktrate überproportional viel mehr Wärme und damit auch Energieverlust produzieren, vgl. [8]. Trotz der Beschränkung von Taktraten steigt die Anzahl von Transistoren in CPUs, durch technologischen Fortschritt, weiter nach dem Moor'schen Gesetz. Das Moor'sche Gesetz besagt, dass sich die Anzahl von Transistoren innerhalb der CPUs alle 12 bis 18 Monate verdoppelt, vgl. [5]. Statt die Laufzeiten in aktuellen CPUs zu verringern, und damit die Anzahl der Transistoren durch eine weitere Erhöhung der Taktraten zu realisieren, haben die CPU-Hersteller die Anzahl der Kerne innerhalb der CPUs erhöht. Der Trend, die Anzahl der Kerne in CPUs zu erhöhen, wird sich weiter fortsetzen und der Softwareentwicklung werden bald eine Vielzahl von CPUs zur Verfügung stehen, vgl. [9]. Die Softwareentwicklung wird sich dem Trend der Mehrkernsysteme zukünftig anpassen müssen und in den nächsten Jahren vermehrt die nebenläufige Programmierung einsetzen, vlg. [7]. Die grundsätzlichen Mechanismen zur nebenläufigen Programmierung sind bei den meisten imperativen Sprachen die gleichen. Mutexe und Semaphore zur Synchronisation einer einzelnen Ressource oder einer Ressourcengruppe.

Der Living Place Hamburg ist ein Projekt der Hochschule für Angewandte Wissenschaften(HAW) Hamburg zur Integration aktueller Technologien in das tägliche Leben. Der Living Place selbst ist eine Wohnung die von Studenten zur Verwirklichung von Projektarbeiten zu ihren Forschungsthemen bietet. Diese Wohnung liegt auf dem Campus Berliner Tor und stellt das Zentrum des Projektes 1 im folgenden Semester(SS10) dar. Im Kontext dieses Projektes werden Sensoren und Aktoren in die Wohnung integriert und liefern Daten zur Verarbeitung an einen oder mehrere Server. Diese müssen in der Lage sein diese Daten nebenläufig zu verarbeiten. Im Laufe der Vorarbeiten zum Anwendungen(AW) 1 Seminar wurde sich mit der Programmiersprache Clojure auseinandergesetzt, vgl. [2]. Diese Sprache bietet eine alternative Lösung zur Programmierung von Anwendungen mit Anforderungen an die nebenläufige Datenverarbeitung. Die Konzepte dieser Sprache werden in dieser Seminarausarbeitung gezeigt und im folgenden Projekt eingesetzt, um einen hohen Grad an Nebenläufigkeit in die Verarbeitung und Abstraktion von Sensordaten zu bringen.

Im 2. Kaptiel werden die Schwachstellen der aktuellen Programmiersprachen im Umgang mit mehreren CPUs erläutert und die dabei entstehenden Konflikte verdeutlicht. Hierzu wird auch die Programmiersprache Clojure im Gegensatz zu konventionellen imperativen Sprachen herangezogen. Das 3. Kapitel setzt sich mit den Grundlagen der Sprache auseinander und erläutert die Synchronisationsmechanismen, die es ihr erlauben nebenläufiges Programmieren zu vereinfachen. Den Abschluss dieser Ausarbeitung bietet eine

Zusammenfassung mit Ausblick auf den Einsatz der Programmiersprache Clojure im Living Place Hamburg.

2 Nebenläufigkeitsmechanismen in der Softwareentwicklung

2.1 Grundlagen

Die Programmierung von nebenläufigen Anwendungen bietet viele Herausforderungen, die sich bei sequentiellen Anwendungen nicht ergeben. Generell gibt es zwei Kategorien von nebenläufigen Anwendungen. In der einen Kategorie ist das Ziel, die Arbeit zu teilen die mit der Verarbeitung von einander unabhängiger Daten verbunden ist. Die Instruktionen zur Verarbeitung der Daten werden nebenläufig ausgeführt um eine schnellere Verarbeitung zu gewährleisten. In der anderen Kategorie ist das Ziel, die Koordination zwischen der Ausführung von Instruktionen, zur Verarbeitung von einander abhängiger Daten, zu koordinieren. In beiden Fällen müssen die Daten vor nebenläufigen Zugriffen geschützt werden, um die Korrektheit des Ergebnisses zu gewährleisten. Die meist verwendeten Synchronisationsmechanismen sind Locks, Actors und Software Transactional Memory(STM), vgl. [10].

Locks Lockbasierte Nebenläufigkeit bietet Threads einen Mechanismus für sichere Zugriffe auf eine Ressource. In der einfachsten Form bedeutet das den exklusiven Zugriff. Alle anderen Threads die eventuell auch auf diese Ressource zugreifen wollen warten bis der Thread der das Lock besitzt es wieder frei gibt. Der Zugriff auf eine Ressourcengruppe wird meist auch mit einem Lock-Mechanismus synchronisiert, wobei hier Absprachen eine Reihenfolge des Lockings bestimmen. Die Programmierer in einem Entwicklungsteam müssen diese Reihenfolgen festlegen und befolgen. Es gibt keine erzwungenen sprachinternen Richtlinien, um einer falsch angewandten Reihenfolge vorzubeugen. Der Speicher, für die Ressourcen auf die die Threads zugreifen, wird von allen Threads geteilt.

Actors Das Aktor-Modell basiert auf dem Prinzip der Nachrichtenkommunikation. Aktoren sind Softwareentitäten die als leichtgewichtige Threads oder Prozesse Instruktionen ausführen und mittels Nachrichten kommunizieren. Aktoren senden und empfangen Nachrichten um neue Aktoren zu erstellen, Nachrichten an andere Aktoren zu senden oder verarbeiten empfangene Nachrichten. Aktoren teilen sich den Speicher nicht, so müssen Nachrichten die zu verarbeitenden Daten enthalten und können u.U. bei großen Datenstrukturen hohe Übertragungszeiten produzieren. Alle Daten auf denen Aktoren Berechnungen durchführen, können nur die Aktoren selbst ändern, vgl. [10].

Software Transactional Memory Der Software Transactional Memory Ansatz stellt eine Alternative zu den o.g. Ansätzen dar, indem er nebenläufigen Threads Zugriffsmechanismen für Shared Ressources bietet. Um eine Shared Ressource zu manipu-

lieren, werden Transaktionen eingesetzt. Diese Transaktionen verändern indirekte Referenzen auf die zu manipulierenden Daten. Nur in den Transaktionen können Änderungen an Daten vorgenommen werden. Durch diesen Mechanismus wird der Programmierer dazu gezwungen, Änderungen in Codeblöcken zu beschreiben die durch automatische Mechanismen dem Fehlverhalten des Programms vorbeugen können. Die Basis dieser Technologie bietet die aus dem Umfeld der Datenbanken bekannte ACID Terminologie, vgl. [3]:

A steht für atomic. Atomar bedeutet in diesem Kontext, dass entweder alle Transaktionen erfolgreich oder keine durchgeführt werden.

C steht für consistent. Konsistent bedeutet, dass alle Daten die an einer Transaktion beteiligt sind, am Ende und Anfang der Transaktion konsistent sind.

I steht für isolated. Isoliert bedeutet, dass alle Änderungen die innerhalb einer Transaktion vorgenommen werden, nur in dieser Transaktion sichtbar sind, bis diese abgeschlossen ist.

D steht für durable. Dauerhaft bedeutet, dass alle in einer Transaktion vorgenommenen Änderungen dauerhaft übernommen werden, selbst wenn es einen Fehler in Netzwerk oder Hardware gibt. In der Software Variante der ACID Terminologie, wird dieses Kriterium nicht eingesetzt, denn in dem Fall wäre eine Datenbank zu nutzen.

Insbesondere bei der Programmierung mit imperativen Sprachen, wie z.B.: C, C++, Java oder C#, werden Locks wegen des hohen Verbreitungsgrades eingesetzt. Die Herausforderungen die sich dabei ergeben entstehen durch fehlende Absprachen und Komplexität der Aufgabenstellung. Sollte sich einer der beteiligten Programmierer nicht an die Reihenfolge des Lockings halten, so kann es zu Deadlocks(Verklemmungen) kommen. Führt man die Priorisierung von Threads ein, kann es zu Starvation(Verhungern) kommen. Man kann diesen Problemen mit Namenskonventionen entgegenwirken, aber die Problematik des manuellen Lockings bleibt bestehen. Aktorsysteme wurden weitestgehend auch in imperativen Sprachen implementiert, sodass dieser Mechanismus zur nebenläufigen Programmierung auch in diesen Sprachen eingesetzt werden kann. Die Implementationen der Aktorsysteme sind meist Bibliotheken die vom Programmierer benutzt werden können, aber nicht Teil der Sprache selbst.

Clojure bietet zur Lösung dieser Problematik eine Kombination aus Agent- und Software Transactional Memory- System an. Es gibt in Clojure keine Möglichkeit, auf Shared Ressources (schreibend) zuzugreifen, ohne die vorhandenen Synchronisationsmechanismen zu nutzen. Die Sprache selbst implementiert alle notwendigen Mechanismen zur Programmierung nebenläufiger Anwendungen und erlaubt darüber hinaus keinem Programmierer diese nicht zu nutzen. Absprachen und Konventionen zur Sicherung von Shared Ressources werden damit überflüssig und das beschleunigt den Entwicklungsprozess.

3 Clojure

Clojure ist eine funktionale, dynamische Programmiersprache die auf der Java Virtual Machine(JVM) und der CLR(Microsoft Common Language Runtime) läuft. In dieser Ausarbeitung wird Bezug auf die Implementierung auf der JVM genommen. Sie kombiniert interaktive Entwicklung wie bei Scriptsprachen mit einer Effizienten und robusten Infrastruktur zur nebenläufigen Multithread-Programmierung. Dabei verteilt die JVM, die zur Verarbeitung von nebenläufigen Instruktionen nötigen Threads, auf die ihr zur Verfügung stehenden CPUs. Clojure Programmcode wird direkt in JVM-Bytecode übersetzt, wobei die Funktionalität der Sprache zur Laufzeit verfügbar bleibt. Clojure bietet die Möglichkeiten für direkte Zugriffe auf Java Bibliotheken und optionale Typhinweise um Java-Reflections zu vermeiden. Clojure ist außerdem ein Lisp Dialekt und bietet ein Macro System sowie persistente unveränderliche Datenstrukturen. Wenn veränderliche Datenstrukturen benötigt werden, z.B. für Multithreadsysteme, sichert Clojure den Umgang mit den Shared Ressources über ein Software Transactional Memory- und Agentensystem. Sie wurde von Rich Hickey entwickelt und seit ihrem ersten offiziellen Release[1] stetig erweitert[2], vgl. [2].

3.1 Clojure Grundlagen

Im Folgenden wird eine Auswahl von Clojure Eigenschaften als general-purpose Programmiersprache gezeigt und erläutert, vgl. [3].

Funktional Clojure ist eine, nicht reine, funktionale Programmiersprache und ein LISP(LISt Processing) auf der JVM. Funktionale Programmiersprachen ermöglichen die Behandlung von Funktionen als „first-class“ Objekte. Das Bedeutet, das Funktionen zur Laufzeit erstellt, übergeben und zurückgegeben werden können. Eine Funktion ist dabei ein Datentyp der keinen Unterschied zu anderen Datentypen aufweist. Daten in Clojure sind unveränderlich. Funktionen sind pur, d.h. sie haben keine Seiteneffekte. In Clojure können Funktionen, anders als in rein funktionalen Sprachen(z.B. Haskell), Seiteneffekte haben. Es ist üblich diese Bereiche mit einem *(**do** ...)*-Block zu kennzeichnen.

[1] Clojure 1.0, 04. Mai 2009 - http://clojure.blogspot.com/2009/05/clojure-10.html
[2] Clojure 1.1, 31. Dezember 2009 - http://clojure.blogspot.com/2009/12/clojure-11-release.html

REPL Die REPL (Read Eval Print Loop) ist eine interaktive Programmierumgebung. Sie ermöglicht dem Programmierer die Interaktion mit einer laufenden Anwendung. Der Benutzer einer REPL kann Instruktionen in die REPL eingeben, diese werden dann vom „Reader“ gelesen und in eine Form übersetzt, die vom „Evaluator“ verstanden wird. Der „Evaluator“ wertet die Instruktionen aus und berechnet das Ergebnis der Instruktionen, welches dann vom „Printer“ an den Benutzer zurückgegeben wird. Die „Loop“ steht für eine vom Benutzer zu unterbrechenden Schleife die auf Benutzereingaben wartet, um den o.g. Prozess zu durchlaufen. Ursprünglich wurde sie in der LISP Gemeinschaft etabliert und dann auf verschiedene Sprachen(MATLAB, Ruby, Python, Haskell u.v.m.) übertragen. Ihre Funktionalität gleicht einem Kommandozeileninterpreter für eine Programmiersprache. Es können Programme in die REPL geladen werden und zur Laufzeit konfiguriert oder erweitert werden.

Makros Makros in Clojure ermöglichen dem Programmierer die Erweiterung der Sprache. In den meisten Programmiersprachen erweitert man die Funktionalität der Sprache in der Sprache selbst. Man programmiert also eine Funktion für eine Aufgabe in der Sprache die man verwendet. Clojure hat, wie Lisps, ein Makrosystem. Markos ermöglichen es dem Programmierer die Programmiersprache um Sprachkonstrukte zu erweitern.

Bindings und Namespaces In Clojure werden Bindings verwendet um Daten an Namen zu binden. Das kann auf Verschiedenen Ebenen geschehen. Zum einen auf globaler und threadlokaler Ebene, zum anderen auch auf formlokaler Ebene, wie z.B. in Funktionsaufrufen. Clojure unterstützt das sog. „destructuring“. Dabei werden Argumente als Funktionsparameter bei der Übergabe eines evtl. großen Datums auf den benötigten Bereich destrukturiert, d.h. nur der geforderte Teil der Datenstruktur wird lokal gebunden. Namespaces ermöglichen es dem Programmierer seine Funktionen in Funktionsbereiche oder Anwendungsgebiete zu unterteilen.

Metadaten Metadaten sind Daten die Daten beschreiben. In Clojure ist der Vergleich von zwei Daten ohne*(= x y)* und mit*(**identical?** x y)*s Berücksichtigung der Metadaten möglich. Metadaten werden in Clojure benutzt um Funktionsdokumentationen und Typen zur Optimierung anzugeben.

Java Interoperabiliät Clojure ermöglicht die Nutzung aller Java-Bibliotheken durch eine direkte Möglichkeit Java Klassen oder Instanzen zu erstellen und zu nutzen. Clojure bietet im Umgang mit Java Interoperabiliät viele Macros und Funktionen um den Umgang mit Java effektiver zu gestalten. Zum Beispiel das *(**doto** <class-or-instance> & <member-access-forms>)*-Makro, ruft eine oder mehrere Membermethoden einer Klasse oder Instanz auf, ohne das diese jedes Mal, so wie in Java, ausgeschrieben werden muss.

Lazy-Evaluation Lazy-Evaluation bedeutet, dass ein Datum erst berechnet wird, wenn es verwendet wird. Clojure bietet die Möglichkeit der Programmierung von sog. „lazy-sequences“ durch das *(**lazy-seq** & <body>)*-Makro, welches den <*body*> erst ausführt wenn es nötig ist. Des Weiteren werden alle Zwischenergebnisse gecached, sodass jeder Aufruf eines Zwischenergebnisses, zu sofortigem Ergebnis führt. Durch dieses aus der Funktionalen Programmierung bekannte Paradigma ist es möglich potentiell unendliche Mengen zu definieren, vgl. [11]. Verwendet man ein Datum

aus dieser Menge wird die Menge berechnet bis das Datum in der Menge enthalten ist, vgl. [3].

Sequences Sequences in Clojure abstrahieren den Zugriffsmechanismus für Strings, Listen, Vektoren, Maps, Sets und Bäume. Alle Java-Collections können über diese Abstraktion angesprochen und mit Clojure eigenen Datenstrukturen kombiniert werden. Die drei Zugriffsmethoden die diese Datenstrukktur bildet sind *(**first** <seq>)*, *(**rest** <seq>)* und *(**cons** <elem> <seq>)*. Diese Zugriffsmethoden werden durch typspezifische Funktionen erweitert, eine Liste z.B. bietet auch *(**peek** <col>)* und *(**pop** <col>)* als Zugriffsmethoden. Diese Abstraktion ermöglicht das schreiben von Funktionen mit einem höheren Abstraktionslevel.

3.2 Persistente, unveränderliche Datenstrukturen

In Clojure sind alle Datenstrukturen unveränderlich und persistent. Unveränderlich bedeutet in diesem Kontext, dass ein Datum nie verändert wird, sondern aus ihm ein neues berechnet wird. Persistent bedeutet, dass die Vorgängerversion des Datums bestehen bleibt bis sie nicht mehr Referenziert wird. Durch die Implementierung in Java wird der Aufwand für das Bereinigen des Speichers von der Garbage Collection(GC) geleistet. Alle Datenstrukturen in Clojure erfüllen diese beiden Kriterien, wobei sie ihre Leistungseigenschaften behalten. Die Implementierung der HashMap in Clojure wird im Folgenden beispielhaft beschrieben.

HashMaps zeichnen sich vor allem durch konstante Zugriffszeiten aus. Über einen Key wird, mittels einer Hash- Funktion, in konstanter Zeit auf ein Value zugegriffen. Die Implementierung in Clojure wird über bit-partitioned hash trees(BPHT) mit den o.g. Eigenschaften realisiert, vgl. [1]. Diese BPHTs haben einen Verzweigungsfaktor(branchingfactor) von 32. Dazu wird die HashTable in 5 Bit große Abschnitte unterteilt, vgl. Abb. 3.1 (oben). Jeder dieser Bereiche dient der Unterscheidung von Hashwerten der Daten die in dieser Datenstruktur abgelegt werden sollen. Um ein Datum in einen BPHT einzuspeisen wird der Hashwert des Datums gebildet und in dem Baum an der ersten Stelle abgelegt, in dem er sich abschnittsweise von allen anderen unterscheidet, vgl. Abb. 3.1 (unten).

Wird ein Datum in der HashMap „verändert“, also eine neue HashMap erstellt die sich in einem Datum unterscheidet, so wird das sog. path copying angewendet um die Persistenz zu gewährleisten, vgl. Abb. 3.2. Um die Leistungseigenschaften bei zu behalten, wird nicht der ganze Baum kopiert, sondern nur der Pfad zum neuen Datum im Baum, auch „structural sharing“ genannt. Die Referenzen aller Knoten auf dem Weg werden kopiert und das neue Datum in die Struktur eingefügt. Alle Daten die auf dem Weg zum neuen Knoten durch andere Knoten referenziert werden, bleiben unverändert, sodass ausschließlich die Knoten kopiert werden müssen, die diese Daten referenzieren. Dieses Vorgehen gewährleistet bei einer Baumtiefe von maximal sieben, die Konstante Zugriffszeit, vgl. [3].

Unveränderlichkeit und Persistenz werden durch dieses Zugriffsverhalten realisiert. Ältere Versionen der Datenstruktur bleiben erhalten und können weiterhin referenziert werden solange das Datum benötigt wird. Die referenzierte Version ändert sich nicht, es gibt nur

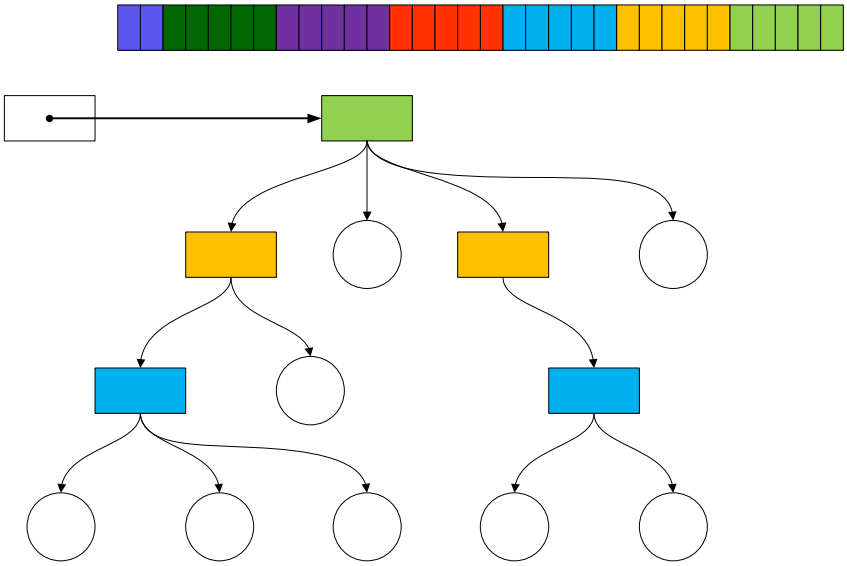

Abb. 3.1: HashMap als bit-partitioned hash tree; Oben: 5 Bit Aufteilung zur Unterscheidung von Elementen; Unten: Repräsentation als Baumstruktur

aktuellere Versionen der Datenstruktur[3]. Werden alte Versionen der Datenstruktur nicht mehr benötigt, so werden sie von der GC der JVM beseitigt.

3.3 Synchronisationsmechanismen

Clojures Synchronisationsmechanismen ermöglichen das konsistente Ändern der Datenstrukturen. Wenn eine Datenstruktur nicht über eine Referenz angesprochen wird ist sie unveränderlich. Das indirekte Referenzieren ermöglicht den Einsatz eines Multi Version Concurrency Control(MVCC) Systems, mit dem der STM in Clojure integriert wird. Das Konzept von MVCC basiert auf Zeitstempeln. Zeitstempel sind diskrete, fortlaufende Zeitpunkte(Nummern) abhängig von der MVCC Umgebung. Eine ändernde Transaktion kann nur Erfolg haben, wenn ihr Zeitstempel kleiner ist als alle anderen Transaktionen die auf das selbe Datum zugreifen. Jedes Datum hat einen Lesezeitstempel, um zu gewährleisten, dass eine Transaktion nicht neuer ist als das letzte gelesene Datum. Sollte das der Fall sein, so wird die Transaktion wiederholt. Der Vorteil dieser Implementation ist, das kein lesender Zugriff blockierend ist. Um in Clojure schreibend auf Daten zuzugreifen, werden vier Referenztypen angeboten, Var, Atom, Agent und Ref. Im Folgenden werden diese Referenztypen erläutert und ihre Zugriffsmechanismen gezeigt, vgl [4] [3].

3.3.1 Var

Vars sind Bindings die zwischen allen Threads geteilt werden, aber auch Threadlokale Daten enthalten können. Sie werden meist für Konstanten verwendet und sollten nicht modifiziert werden. Vars werden wie folgt verwendet:

[3] http://clojure.blip.tv/ - Clojure Concurrency

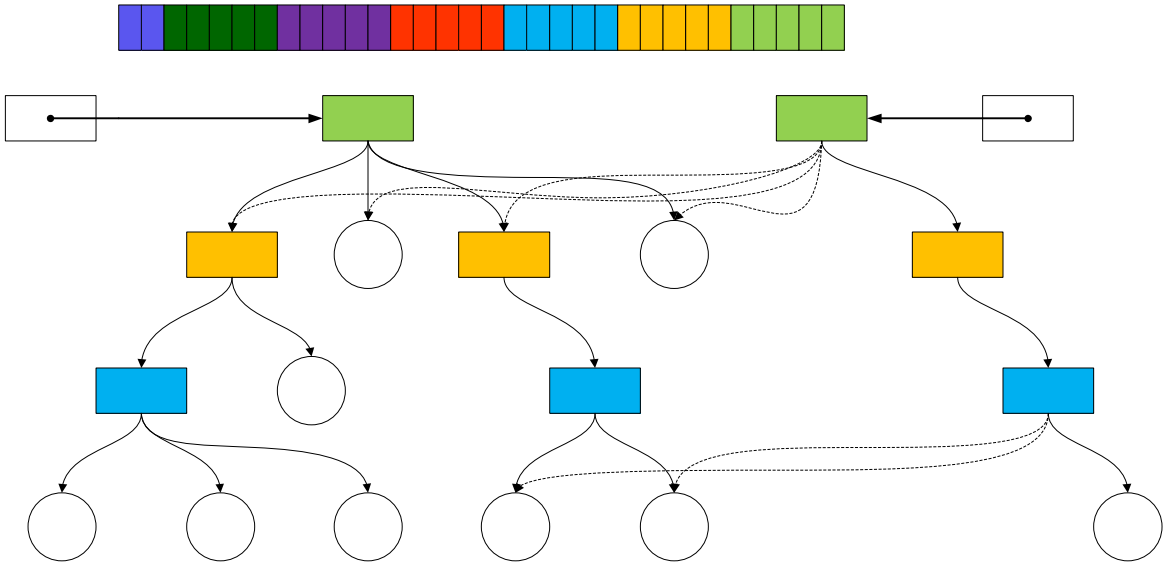

Abb. 3.2: HashMap mit einem zusätzlichen Datum

```
;; create a var
(def <name> <value>)
;; set a var
(set! <name> <value>)
;; bind a local value to a var
(binding [<bindings>] & <body>)
```

3.3.2 Atom

Atoms werden für unkoordinierte, synchrone Updates benutzt. Synchron bedeutet in diesem Kontext, dass der Aufruf einer Funktion zum ändern eines Atoms, erst zurückkehrt, wenn die Änderung vorgenommen wurde. Atome sind für das updaten eines einzelnen Datums. Sie werden eingesetzt um von mehreren Threads aus einzelne Daten zu manipulieren, sodass alle Threads einen sicheren Zugriff auf diese Ressource haben und erlauben das Ändern nur über die folgenden Funktionen:

```
;; create an atom
;; options? could be validator functions or metadata
(atom <initial-state> <options?>)
;; reset an atom
(reset! <atom> <value>)
;; lower level reset, if atom is oldval, it is changed to newval
(compare-and-set! <atom> <oldval> <newval>)
;; updates an atom by calling a function f with arguments args
(swap! <atom> <f> & <args>)
;; dereference the atom
(deref <atom>)
@<atom>
```

3.3.3 Agent

Agenten werden für unkoordinierte, asynchrone Updates verwendet. Agenten werden eingesetzt wenn eine Transaktion sofort zurückkommen soll. Die eigentliche Aufgabe wird

von einem Agent erst in der Zukunft ausgeführt. Folgende Funktionen stehen im Umgang mit Agents zur Verfügung:

```
;; create an agent
;; options? could be validator functions or metadata
(agent <initial-state> <options?>)
;; sends an agent an update function f with arguments args to execute
(send <agent> <f> & <args>)
;; sends an agent an update function f that might block for some time
(send <agent> <f> & <args>)
;; wait for an agent to complete his work
(await & <agents>)
;; wait for some time for an agent to complete his work
(await-for <timeout-ms> & <agents>)
;; dereference an agent
(deref <agent>)
@<agent>
```

3.3.4 Ref

Ref ist der komplexeste Synchronisationsmechanismus in Clojure. Er dient der koordinierten und synchronen Ausführung von Funktionen auf gemeinsam genutzten Variablen. Die Manipulation von Refs kann nur in Transaktionen stattfinden, sodass jede Funktion die eine Ref ändert mit einem *(**dosync** ...)*-Transaktionsblock umschlossen werden muss. Funktionen in einem Transaktionsblock werden sequentiell in der Reihenfolge ausgeführt in der sie ausformuliert werden. Sollte ein Teil der Transaktion fehlschlagen, wird die ganze Transaktion wiederholt. Durch die o.g. Zeitstempel und die Persistenz aller Datenstrukturen wird gewährleistet das Transaktionen Daten bearbeiten können während andere Threads diese lesen. Sobald eine Transaktion abgeschlossen ist, wird der neue Zustand aller beteiligten Bindings konsistent und atomar sichtbar. Im Folgenden die Clojure Funktionen für den Umgang mit Refs:

```
;; create a ref
(ref <value>)
;; create a transition block in that you can modify refs with functions
(dosync & <fs>)
;; change where a reference points to
(ref-set <ref> <new-value>)
;; applys a funtion f with its arguments args to a ref
(alter <ref> <f> & <args>)
;; the order of functions in a commute block must be commutative
;; gives the STM the ability to change the call order for performance
(commute <ref> <f> & <args>)
;; dereference a ref
(deref <ref>)
@<ref>
```

4 Zusammenfassung

Clojure bietet eine Vielzahl von Synchronisationsmechanismen mit denen die Entwicklung von nebenläufigen Anwendungen vereinfacht wird. Im Projekt 1 werden Mechanismen zur Abstraktion von Sensorinformationen im Living Place untersucht. Die in diesem Umfeld entstehenden Anforderungen sind hauptsächlich mit nebenläufiger Datenverarbeitung verbunden, da eine Vielzahl von Sensoren zum Einsatz kommen wird. Die Daten der Sensoren sollen auf den Event Heap(vgl. [6]) übertragen werden. Der Event Heap basiert auf dem Publisher Subscriber Prinzip, bei dem Pusblisher Daten zur Verfügung stellen und Subscriber sich für diese Daten anmelden um sie zugesandt zu bekommen. Publisher und Subscriber sind in diesem Kontext Agenten die mit ihren Sensoren Daten liefern und evtl. Daten empfangen um eine Reaktion auszuführen , vgl. Abb. 4.1. Die

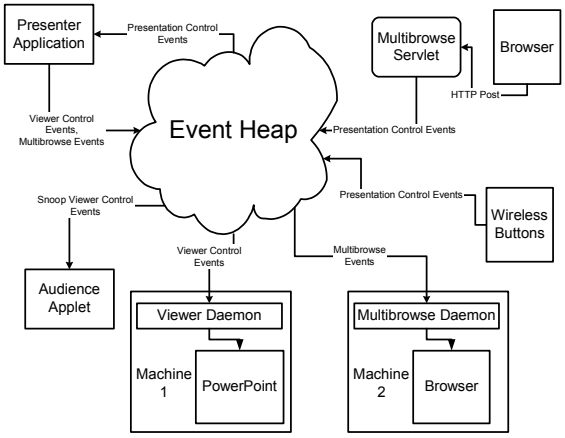

Abb. 4.1: Pfade von Datenströmen zum EventHeap

Problemstellung an dieser Stelle ist zum einen die Menge der Daten, zum anderen die Rechenleistung der Subscriber. Der Event Heap abstrahiert die bereitgestellten Informationen nicht, er dient hauptsächlich als Verteiler der Daten. Sollten mehrere Subscriber für ein Event Interesse zeigen, müssten sie alle die gleiche Abstraktion errechnen. Eine Abstraktion der Sensordaten ist also vor dem Versenden an die Subscriber erforderlich um einen möglichst kleinen Rechenaufwand zu erzielen. Um dieser Problematik vorzubeugen, hat man sich entschieden, eine zentrale Architektur zu verwenden. Die Abstraktion der Daten auf den verschiedenen Abstraktionsleveln erfordert einen hohen Grad an Nebenläufigkeit. Die Abstraktion bedeutet in diesem Fall auch das Zusammenfassen von Informationen wie z.B. Positions- und Bilddaten zu einer bestimmten Person in einem bestimmten Raum.

Mein Beitrag zur Realisierung des Living Place Hamburg, als Forschungsprojekt zur Integration aktueller Technologien in das tägliche Leben, wird sich mit der Bereitstellung

und Abstraktion von Sensordaten auseinandersetzen. Der Event Heap wird mit weiteren Systemen zur Datenverteilung verglichen und evtl. mit einer Abstraktionsfunktionalität erweitert. Auch andere Programmiersprachen, wie z.B. Scala und F#, zur Entwicklung nebenläufiger Anwendungen, werden im Bezug auf Leistungsunterschiede untersucht und verglichen. Des weiteren wird eine Aufgabe sein, die Synchronisationsmechanismen von STM-Systemen in ihrer Implementation zu untersuchen, um die Details der Funktionalität zu durchdringen.

Literaturverzeichnis

[1] BAGWELL, Phil: *Ideal Hash Trees*. 2001. – URL http://lamp.epfl.ch/papers/idealhashtrees.pdf. – Technical Report for the Swiss Institute of Technology Lausanne

[2] CLOJURE.ORG: *Clojure — Language Home*. 2009. – URL http://www.clojure.org. – [Online; Stand 28. Oktober 2009]

[3] HALLOWAY, Stuart: *Programming Clojure*. 2009. – ISBN 978-1-934356-33-3

[4] HICKEY, Rich: *Clojure API for clojure.core*. 2010. – URL http://richhickey.github.com/clojure/clojure.core-api.html. – [Online; Stand 17. Februar 2010]

[5] HOFFMANN S. ; LIENHART, R.: *OpenMP*. 2008. – ISBN 978-3-540-73122-1

[6] JOHANSON, Brad ; FOX, Armando: *The Event Heap: A Coordination Infrastructure for Interactive Workspaces*. 2010. – URL http://graphics.stanford.edu/papers/eheap3/eheap_wmcsa.pdf. – [Online; Stand 18. Februar 2010]

[7] JONES, Simon P.: *Beautiful Concurrency*. S. 385–406. In: *Beautiful Code - Leading Programmers Explain How They Think*, O'Reilly, 2007

[8] RAUBER, Thomas ; RÜNGER, Gudula: *Multicore: Parallele Programmierung*. 2008. – ISBN 978-3-540-73113-9

[9] TOMSHARDWARE.COM: *Intel Demos Single Chip with 48 Cores*. 2009. – URL http://www.tomshardware.com/news/Intel-Cloud-Cores-Processor-CPU,9193.html. – [Online; Stand 2. Dezember 2009]

[10] VOLKMANN, Mark: *Software Transactional Memory*. 2010. – URL http://java.ociweb.com/mark/stm/article.html. – [Online; Stand 14. Februar 2010]

[11] WIKIPEDIA: *Lazy evaluation — Wikipedia, The Free Encyclopedia*. 2010. – URL http://en.wikipedia.org/w/index.php?title=Lazy_evaluation&oldid=345743536. – [Online; Stand 24. Februar 2010]