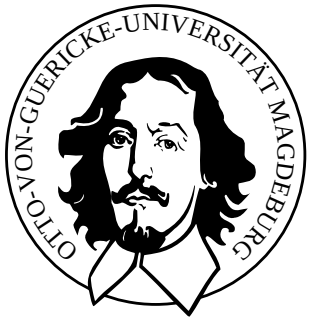
- Diplomarbeit -

Entwicklung eines verteilten Cachesystems für ein geclustertes Ad-Hoc-Netzwerk

Thomas Draband
Mat.Nr.: 154269



„Otto-von-Guericke“ Universität Magdeburg
Fakultät für Informatik
Institut für Verteilte Systeme
Betreuung: Prof. Dr. Edgar Nett
Dipl.-Inf. Daniel Mahrenholz

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

Drahtlose Netzwerkkommunikation ist ein Thema, dass in kurzer Zeit viel Aufmerksamkeit und eine breite Akzeptanz erlangt hat. Die fortschreitende Entwicklung mobiler Geräte, wie Laptops, PDAs und auch Handys stärkt diese Akzeptanz immer weiter. Solch drahtlose Netzwerke, auch WLANs genannt, sind mit dem heutigen Stand der Technik nicht mehr wegzudenken. Hotels ohne WLAN mit der Möglichkeit darüber auf das Internet zuzugreifen, haben bei dem Kundenstamm der Geschäftsleute keine guten Chancen. In naher Zukunft wird es in Ballungszentren wie Bahnhöfen und Einkaufszentren wahrscheinlich kein Problem sein, mit einem PDA oder Laptop drahtlos ein Produkt bei eBay zu suchen. WLANs werden heute hauptsächlich eingesetzt, um drahtlos mit zentralisierten Diensten in LANs oder dem Internet zu kommunizieren. Diese Anwendung drahtloser Netze wird als Infrastrukturnetzwerk bezeichnet. Die Basis dafür sind immobile Zugangspunkte, sogenannte Access Points, zu drahtgebunden Netzen. Allerdings können drahtlose Netzwerke auch ohne Infrastruktur existieren. Diese werden dann als Ad-Hoc Netze bezeichnet. Hier wird die Möglichkeit ausgenutzt, dass mobile Endgeräte direkt ohne äussere Steuerung miteinander kommunizieren können. Probleme bereitet die Tatsache, dass in Infrastrukturnetzen Dienste wie Namensdienste, Gateways und Authentifizierungsserver existieren, die das vorhandene Netz transparent machen und eine effektive Nutzung ermöglichen. Diese Dienste sind bei einem Ad-Hoc-Netz nicht verfügbar.

Es gibt zwei Möglichkeiten, in Ad-Hoc-Netzwerken Informationen über das Netz zu erhalten. Zum einen können Knoten nach bestimmten Informationen suchen, in dem sie das gesamte Netzwerk mit ihrer Anfrage fluten und auf eine positive Antwort hoffen. Zum anderen werden durchgängig Netzwerkinformationen in Paketen übertragen, die ein Knoten entweder nur passiv mithört oder an andere Knoten weiterleitet. Die Flutung ist eine sehr teure Übertragung, da unabhängig von der Grösse des Netzes versucht wird, alle Knoten zu erreichen. Je größer ein Netzwerk wird, je mehr Knoten führen potentiell Flutungen aus. Die Bandbreite des Netzes wird weiter ausgelastet. Im schlimmsten Fall ist das Netzwerk nur noch mit den Flutungen beschäftigt. Die zweite Möglichkeit, das passive Mithören, setzt eine Speicherung der Informationen auf den Knoten voraus. Dies lässt sich durch einen Cache unproblematisch realisieren, ohne dass der benötigte Speicher zu große Dimensionen annimmt. In diesem Cache können dann auch die Ergebnisse von Flutungen zwischengespeichert werden. Die Informationen können dann wiederverwendet werden. Eine Reduktion der Flutungen ist das Ergebnis. Ungünstig ist aber weiterhin, dass andere Knoten mit den gesammelten Informationen eines anderen Knotens nicht arbeiten können.

Ziel dieser Diplomarbeit ist es, ein verteiltes Cachesystem zu entwickeln, dass es ermöglicht, die gesammelten Informationen einzelner Knoten im gesamten Netzwerk zu nutzen. Auf Grund des Einsatzes in Ad-Hoc-Netzwerken muss es fehlertolerant

sein, um mit der Mobilität der Knoten umgehen zu können. Redundanz der Daten sowie eine selbstorganisierende Struktur müssen existieren.

1.2 Einbettung

Das Umfeld der Diplomarbeit ist durch das Projekt „Publisher/Subscriber Middleware“ der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ festgelegt. Ziel der Middleware ist die Bereitstellung einer Umgebung, die oberhalb der Netzwerkschicht Kommunikation mit Dienstgütegarantien in geclusterten Ad-Hoc-Netzwerken ermöglicht. In Systemen, die diese Anforderungen nicht bereits in der Netzwerkschicht erfüllen, ist mit Hilfe der Middleware dann echtzeitfähige Netzwerkkommunikation möglich. In Kapitel 2.7 wird die Middleware genauer beschrieben.

Für die Entwicklung der Middleware wurde zuvor die GEA-Schnittstelle entwickelt, die es ermöglicht, die Implementierung unabhängig von der Plattform zu realisieren. Sie stellt grundlegende Primitive für die Entwicklung ereignisbasierter Netzwerkprotokolle zur Verfügung. Die Middleware basiert vollständig auf GEA und wird somit ereignisbasiert realisiert. Eine Beschreibung der GEA-Schnittstelle ist in Kapitel 2.6 zu finden.

1.3 Aufgabenstellung

In dieser Diplomarbeit soll ein verteiltes Cachesystem entwickelt werden, dass sich in die „Publisher/Subscriber Middleware“ integriert. Ziel ist es, ein Cachesysetem bereitzustellen, das den Komponenten der Middleware ermöglicht, Daten, die zur Laufzeit des Systems gesammelt werden, zwischenzuspeichern. Dabei sollen diese Daten sich selbstständig im gesamten Netz verbreiten, so dass die gesammelten Informationen von allen Knoten des Netzes verwendet werden können.

Das benötigte Kommunikationsaufkommen des Caches darf die Bandbreite, die der Middleware für die Bereitstellung von Kommunikation mit Dienstgütegarantien zur Verfügung steht, nicht einschränken. Der Cache darf somit nur Best-Effort-Traffic für die eigene Kommunikation benutzen. Best-Effort-Daten werden von der Middleware dabei immer nur übertragen, wenn Bandbreite verfügbar ist, die nicht für andere Übertragungen verplant ist.

Um die Ressourcen der Knoten zu schonen, müssen die Daten des Caches verteilt gespeichert werden. Dabei muss eine faire Verteilung auf die Knoten realisiert werden, um einseitige Belastungen von einzelnen Knoten zu vermeiden. Da es sich bei den Zielnetzen um Ad-Hoc-Netzwerke handelt, ist eine inhärente Dynamik der Netzwerktopologie gegeben. Das System muss in der Lage sein, mit dieser Dynamik umzugehen und trotzdem so effizient wie möglich zu arbeiten. Die Einführung von Redundanzen ist somit grundlegend. Das Cachesystem muss Fehler in der Kommunikation tolerieren. In diesem Zusammenhang ist auch die Konsistenzanforderung, die an die Datenbestände unterschiedlicher Knoten gestellt

wird, zu betrachten. Die Konsistenzanforderung beeinflusst primär die Toleranz, mit der Kommunikationsfehler akzeptiert werden. Basierend auf dieser Fehlertoleranz muss ein Kommunikationsprotokoll entwickelt werden, dass die Einhaltung der geforderten Datenkonsistenz ermöglicht.

Da es sich um einen Cache handelt, altern die Informationen und müssen, wenn die maximale Grösse des Cachespeichers erreicht ist, ersetzt werden. Auf welche Weise die Alterung und die Ersetzung gehandhabt wird, muss betrachtet werden, um ein optimales Verfahren für das Cachesystem zu ermitteln.

Durch die Einbettung des Cachesystems ist die Entwicklungsumgebung vollständig definiert. Als Teil der Publisher/Subscriber Middleware benutzt es nur Funktionen der Middleware und der Schnittstelle zur Entwicklung ereignisbasierter Netzwerkprotokolle, GEA, auf die die gesamte Middleware basiert.

1.4 Ergebnis

Im Rahmen dieser Diplomarbeit wurde ein Cachesystem für den Einsatz in der Publisher/Subscriber Middleware entwickelt. Die Verteilung der Daten basiert auf der Clusterstruktur, die die Middleware bereitstellt. Ein Cluster bildet dabei immer genau eine in sich geschlossene Datendomäne. Clusterübergreifend arbeiten die Cluster dann unabhängig voneinander. Im gesamte Cachesystem werden die Daten per Diffusion von einem Cluster an angrenzende Cluster weitergereicht.

Die Verteilung innerhalb eines Clusters ist flach organisiert. Jede Cacheline ist mit einem eindeutigen Schlüssel ausgestattet. Dadurch ist Unabhängigkeit von den eigentlichen Daten geschaffen. Über eine Hashfunktion werden die Cachedaten anhand ihrer Schlüssel in Restklassen aufgeteilt. Alle Knoten bekommen einen Zuständigkeitsbereich über diese Restklassen zugewiesen. Um die Fairness zu gewährleisten, sind die Bereiche asymmetrisch über die Auslastungen der einzelnen Restklassen verteilt. Jeder Knoten bekommt aber keinen eigenen Bereich, sondern abhängig von Anzahl der Knoten werden Gruppen mit gleichen Bereichen gebildet. Mit steigender Anzahl Knoten steigt auch die Anzahl der Gruppen und gleichzeitig die Anzahl der Replikate bzw. Mitglieder der Gruppen. Bei Verlust von Knoten bis zu einer bestimmten Menge gehen dem Cachesystem somit keine Daten verloren.

Alterung und Ersetzungsstrategie werden auf jedem Knoten einzeln behandelt. Die Daten werden mit Zeitstempeln hinterlegt, so dass sie ab einem bestimmten Alter nicht mehr verwendet werden. Ebenfalls über Zeitstempel ist die Ersetzungsstrategie realisiert. Sie arbeitet nach dem Least-Recently-Used-Verfahren. Dabei werden immer die Daten ersetzt, die schon am längsten nicht mehr abgefragt wurden.

1.5 Aufbau

In Kapitel 2 werden alle nötigen Grundlagen, die zum Verständnis dieser Arbeit notwendig sind, behandelt. Wichtige Punkte sind hier Ad-Hoc-Netzwerke im Allgemeinen und die spezialisierte Form der geclusterten Ad-Hoc-Netzwerke, wobei

Letzteres das Umfeld des Cachesystems darstellt. Die beiden Kommunikationsmodelle Client/Server und Publisher/Subscriber werden vergleichend betrachtet, um die Eigenschaften des Umfeldes, das auf dem Publisher/Subscriber Modell basiert, genauer zu erörtern. Durch seine Popularität und gegensätzliche Zielstellung bietet sich das Client/Server Modell zur Verdeutlichung der Eigenschaften des Publisher/Subscriber-Modells im direkten Vergleich an. GEA, die Schnittstelle zur ereignisbasierten Programmierung des Zielsystems, wird ebenso vorgestellt. Relevant sind hierbei die Ziele, die mit diesem Konzept für die Entwicklung von Netzwerkprotokollen erreicht werden.

Kapitel 3 beschäftigt sich mit dem Konzept des Cachesystems. Beim Design werden mehrere Gesichtspunkte, abhängig vom Umfeld des Systems, beachtet. Die Möglichkeiten und Probleme eines verteilten Cachesystems in geclusterten Ad-Hoc-Netzwerken werden diskutiert. Die Diskussion führt zu einem Designentschluss, der später als Teil dieser Diplomarbeit umgesetzt wird. Zuerst wird die Verteilung untersucht. Anschliessend wird die Alterung und die damit verbundene Ersetzungsstrategie sowie die Verbreitung und Synchronisation der Daten diskutiert. In einer stochastischen Analyse wird die Effektivität theoretisch ermittelt.

Die Struktur der Implementierung sowie die technischen Lösungen einiger Probleme sind in Kapitel 4 beschrieben. Die Implementierung des Cachesystems wird dabei Schritt für Schritt von der Speicherstruktur auf einem Knoten bis zur Diffusion der Daten in benachbarte Cluster beschrieben.

2 Grundlagen

2.1 Begriffe

Single-Hop-Netzwerk: Bei Single-Hop-Netzwerken können alle Knoten direkt miteinander kommunizieren. In drahtlosen Netzwerken wird diese Eigenschaft durch die Reichweiten der Sender und Empfänger definiert. Alle Knoten müssen dafür gegenseitig in Reichweite liegen, so dass jeder einzelne Knoten mit allen anderen kommunizieren kann.

Multi-Hop-Netzwerk: Bei Multi-Hop-Netzwerken ist die direkte Kommunikationsmöglichkeit wie in Single-Hop-Netzwerken nicht gegeben. Es ist möglich, dass einzelne Knoten mit bestimmten anderen Knoten des Netzwerkes nur durch Hilfe anderer Knoten kommunizieren können. Dabei leiten Knoten die Pakete anderer Knoten weiter. Ein drahtloses Multi-Hop-Netzwerk besteht aus einer Menge von Knoten, die sich im Gegensatz zu Single-Hop-Netzwerken, nicht alle gegenseitig in Reichweite befinden. Durch die Reichweiten einzelner Knoten ist das gesamte Netzwerk vollständig verbunden, so dass keine Partitionierungen vorliegen. Nun gibt es Kommunikationsverbindungen, die nur durch die Weiterleitung unbeteiligter Knoten möglich sind.

Cacheline: Eine Cacheline ist eine Dateneinheit im Cache. Sie repräsentiert einen Eintrag im Cache.

Hit: Ein Hit ist in der Terminilogie von Caches eine erfolgreiche Anfrage an den Cache, die durch das Auffinden einer Cacheline gekennzeichnet ist.

Miss: Ein Miss ist das Gegenteil eines Hits. Eine gesuchte Cacheline befindet sich nicht im Cache.

Unicast: 1-zu-1-Übertragungen von einem Sender zu genau einem Empfänger heißen Unicast-Übertragungen.

Multicast: Bei Multicast-Übertragungen werden Pakete von einem Sender zu einer Menge von Empfängern übertragen. Das Ergebnis sind 1-zu-n-Übertragungen.

Best-Effort: Vorgänge, wie Netzwerkübertragungen, die keinen zeitlichen Anforderungen und Einschränkungen unterliegen, werden mit Best-Effort ausgeführt. Das heißt, dass sie je nach Verfügbarkeit der benötigten Ressourcen ausgeführt werden.

Fehlertoleranz: Fehlertoleranz in Systemen ist dann gegeben, wenn die Funktion des Systems nicht durch einzelne Fehler beeinträchtigt wird.

Redundanz: Werden Daten mehr als einmal hinterlegt, sind diese redundant. Redundanz ist ein Mittel, um fehlertolerante Systeme zu entwickeln.

2.2 Ad-Hoc-Netzwerk

Netze ohne Infrastruktur werden **Ad-Hoc-Netzwerke** genannt. In ihnen kann ohne die Existenz von zentralisierten Infrastrukturkomponenten kommuniziert werden. Sie organisieren sich selbst (vgl. [Milanovic] S. 257).

Es lassen sich zwei Typen unterscheiden. Zum einem gibt es Single-Hop-Ad-Hoc-Netzwerke, bei denen analog zu Sprechfunksystemen die Kommunikationspartner nur direkt in Kontakt treten können. Bestimmt wird die Konnektivität durch die Reichweite der Funksender. Beim Idealfall eines vollständig verbundenen Netzes, also einem Netz, bei dem alle beteiligten Knoten gegenseitig in Reichweite sind, sind andere Methoden auch nicht notwendig. Allerdings ist dieser Idealfall eher theoretischer Natur und kommt nur äußerst selten oder bei sehr kleinen Netzen vor. Multi-Hop-Ad-Hoc-Netzwerke können dagegen über ihre Sendereichweiten hinaus kommunizieren. Die Pakete werden dabei nicht nur über direkte Verbindungen weitergegeben, sondern auch zu entfernten Empfängern mittels anderer Knoten übertragen. Die Teilnehmer an einem solchen Netzwerk treten durch Interaktion miteinander in Verbindung. Bei Ad-Hoc-Netzen benötigen die Endgeräte die Funktionalität von Routern, da es im Gegensatz zu Infrastrukturnetzen keine zentralen Router für die Verbindung einzelner Teilnetze gibt (vgl. [HJ2001] S. 61). Das heißt, es bedarf keiner zentralen Administration, und es gibt keine Basisstationen. Ad-Hoc-Netzwerke sind in der Lage, sich selbst zu konfigurieren und entstehen selbstständig, wenn Geräte in Kommunikationsreichweite gelangen. Der größte Vorteil eines Ad-Hoc-Netzes liegt also in der Möglichkeit, ohne externe Hilfe ein Netzwerk zu schaffen (vgl. [Cardei] S. 88).

Eine hervorstechende Eigenschaft der Teilnehmer von Ad-Hoc-Netzwerken ist ihre Mobilität. Immobile Endgeräte lassen sich einfach in Infrastrukturnetze integrieren. Mit mobilen Endgeräten ist das nicht so einfach möglich. Im Allgemeinen kann man davon ausgehen, dass alle Teilnehmer eines Ad-Hoc-Netzes deshalb mobil sind. Einige beispielhafte Vertreter sind dabei Notebooks, PDAs, Roboter oder Sensoren. Dies hat zur Folge, dass die Topologie des Netzes einer ständigen Dynamik unterliegt. Die Kommunikationsbeziehungen unter den Endgeräten, die durch die relative Lage zueinander über die Reichweiten definiert sind, können somit ständigen Änderungen unterliegen. Auch die Menge der Netzwerkknoten ist von der Dynamik betroffen. Es können jederzeit Knoten das Netzwerk verlassen, so wie auch jederzeit welche dazu kommen können.

2.3 Geclustertes Ad-Hoc-Netzwerk

In geclusterten Ad-Hoc-Netzwerken werden die Knoten des Netzes geclustert, so dass eine Struktur entsteht, in der nicht jeder Knoten mit seinen spezifischen Verbindungseigenschaften einen eigenen Kommunikationsknoten darstellt. Ein Cluster ist hierbei dann ein Teilnetz von Knoten, das die Eigenschaften eines Single-Hop-Netzwerkes aufweißt. Die Netzwerktopologie wird dann nicht mehr durch die Verbindungen aller Knoten untereinander, sondern durch die Verbindungen der

Cluster untereinander gebildet.

Die Publisher/Subscriber Middleware, in der das Cachesystem eingesetzt werden soll, clustert das Netzwerk. Die genaue Definition des geclusterten Netzwerkes lautet wie folgt:

Cluster: Ein Cluster wird durch genau einen Knoten definiert. Dies ist der Clusterhead (CH). Jeder Knoten kann die Funktion eines Clusterheads übernehmen. Es ist nur von der Topologie des Ad-Hoc-Netzes abhängig. Alle Mitgliedsknoten des Clusters befinden sich in Sende- und Empfangsreichweite des Clusterheads. Die Anzahl der Clients ist auf 15 beschränkt. Ein Cluster besteht somit aus maximal 16 Knoten. Alle Knoten haben eindeutige Adressen. Die Adresse des Clusterheads identifiziert einen Cluster eindeutig.

Client: Alle Mitgliedsknoten neben dem Clusterhead heißen Clients. Sie können zu mehr als einem Cluster gehören.

Gateway: Ein Gateway (GW) ist ein Client, der zu mehreren Clustern gehört. Die Kommunikation zwischen Mitgliedern unterschiedlicher Cluster geht immer über diese Gateways.

Jeder Knoten eines Clusters ist entweder Client oder Clusterhead. Dabei sind alle Clients in Reichweite des Clusterheads. Die Kommunikation zwischen Clusterhead und Clients ist direkt möglich. Clients müssen untereinander nicht in Reichweite sein, da jegliche Kommunikation innerhalb des Clusters über den Clusterhead geleitet wird. Direkte Client-zu-Client-Kommunikation ist damit nicht möglich. Abbildung 2.3 zeigt den Ausschnitt eins Netzwerkes. Zu sehen sind dort drei Cluster, die über Gateways miteinander verbunden sind.

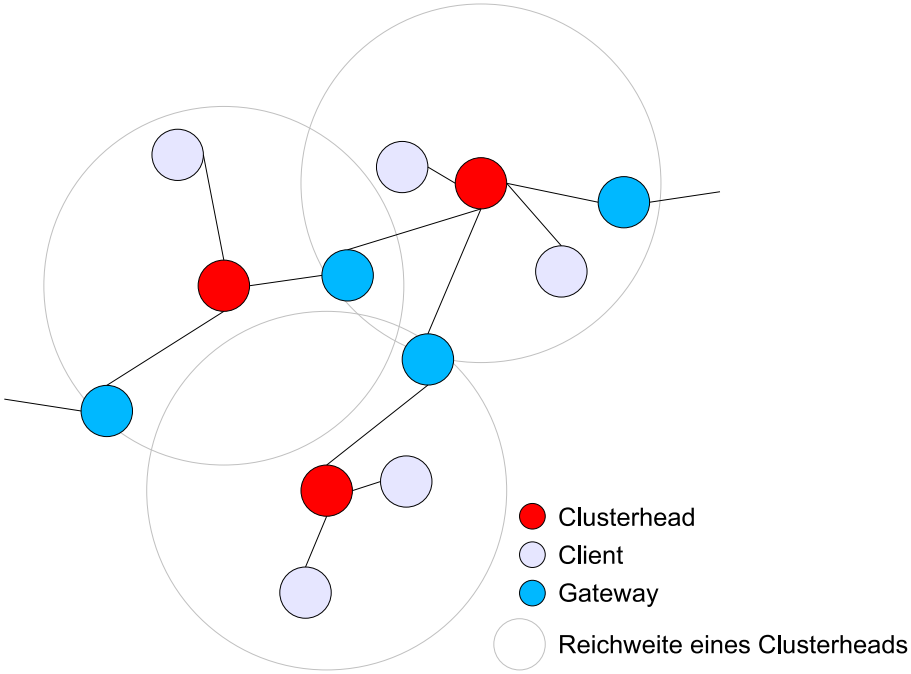


Abbildung 2.3: geclustertes Ad-Hoc-Netzwerk

Die Clients eines Clusters befinden sich immer in Reichweite des zugehörigen Clusterheads. Gateways befinden sich immer in Bereichen, in denen sich die

Reichweiten mehrerer Clusterheads überlappen. Da jegliche Kommunikation im Cluster über den Clusterhead geleitet wird, kann jeder Client einen anderen Client des selben Clusters mit 2 Hops erreichen.

(vgl. [CMANET])

2.4 Verteilte Systeme

Die Voraussetzung für ein verteiltes System ist ein Rechnernetz. In einem Rechnernetz sind mehrere Computer über ein geeignetes Netzwerk miteinander verbunden. Über geeignete Netzwerkprotokolle können die Teilnehmer dieses Netzwerkes miteinander kommunizieren. In verteilten Systemen wird die Kommunikation in Rechnernetzen dazu genutzt, Daten verteilt über verschiedene Knoten im Netzwerk zu verarbeiten. „Bei einem verteilten System arbeiten Komponenten zusammen, die sich auf vernetzten Computern befinden und die ihre Aktionen durch den Austausch von Nachrichten koordinieren“ ([CDK2002] S.17). Anders als bei Rechnernetzen, wo einzelne Knoten auch autark voll funktionsfähig sind, können Knoten eines verteilten Systems nur eingeschränkt oder gar nicht agieren. Dies ist ganz vom Grad der Integration der einzelnen Knoten im verteilten System abhängig. Ein weiterer Unterschied zu Rechnernetzen ist die Transparenz des Netzes. Applikationen arbeiten mit Daten von entfernten Knoten auf die gleiche Art wie mit lokalen Daten. Eine Differenzierung zwischen lokalem und Netzwerkzugriff ist für Anwendungen und Anwender somit nicht möglich.

Ein repräsentatives Beispiel für ein derartiges System ist ein Netzwerk aus Rechnern in einem Unternehmen oder einem Institut. Ziel dieser Netze ist die „[...] computergestützte Zusammenarbeit (CSCW, Computer-Supported Cooperative Working), wobei eine Gruppe von Benutzern zusammenarbeitet und Ressourcen in einer kleinen, abgeschlossenen Gruppe direkt gemeinsam nutzt wie beispielsweise Dokumente“ ([CDK2002]). Zwei häufige Vertreter dieser Systeme sind Microsoft Windows und UNIX Netzwerke. Hauptpfeiler sind hier zentrale Authentifizierung und Bereitstellung von einheitlichen Daten im gesamten Netzwerk.

Die Basis der meisten verteilten Systeme ist das **Client/Server-Modell**. „Der Begriff Server [...] bezieht sich auf ein ausgeführtes Programm (einen Prozess) auf einem vernetzten Computer, das Anforderungen von auf anderen Computern ausgeführten Programmen entgegennimmt, die den Dienst benötigen. Der Server stellt eine geeignete Antwort bereit. Die anfordernden Prozesse werden als Clients bezeichnet“ ([CDK2002]). In Abbildung 2.4 ist ein Netz aus 3 Computern zu sehen. Deutlich wird hier, dass Computer sowohl als Client als auch Server agieren können. Mittelpunkt dieses Modells ist die direkte Adressierung aller Kommunikationen. Im Allgemeinen entstehen dabei 1-zu-1-Verbindungen. Durch die Zentralisierung von Serverprozessen in üblichen Umgebungen hat sich der Begriff Server irritierender Weise für Computer mit zentralen Serverprozessen durchgesetzt. Bei Betrachtungen des Client/Server-Modells kommt es deshalb häufig zu Missverständnissen.

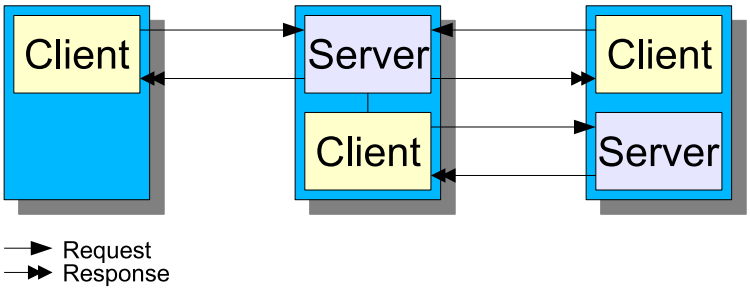


Abbildung 2.4: Client/Server-Modell

Eine Alternative für die Entwicklung von verteilten Systemen ist das **Publisher/Subscriber-Modell**. Wie zuvor lassen sich Prozesse in zwei Klassen unterteilen. Wenn man diese direkt abbilden möchte, ist der Publisher das Äquivalent zum Server und der Subscriber das Äquivalent zum Client. Publisher stellen Daten im verteilten System zur Verfügung. Subscriber empfangen und nutzen diese. Allerdings gibt es zwischen den beiden Modellen grundlegende Unterschiede, weshalb man Publisher/Subscriber nicht als Mutation von Client/Server betrachten darf. Es gibt keine direkte knotenbasierte Adressierung. Das resultiert aus der **inhaltsbasierten Kommunikation**, die das Modell realisiert. Publisher verbreiten bestimmte Inhalte im Netzwerk. Subscriber melden ihr Interesse an bestimmte Inhalte, bzw. bestimmte Klassen von Inhalten, im Netzwerk an. Diese Anmeldung wird als **Subscription** bezeichnet. In einem Netzwerk können mehrere Subscriber und auch Publisher für ein und dieselben Inhalte existieren. Das Netzwerk hat nun die Aufgabe, Daten der Publisher per Multicast an alle angemeldeten Subscriber zu übertragen. Statt einer 1-zu-1-Kommunikation wie beim Client/Server-Modell entsteht so eine n-zu-m-Kommunikation. Abbildung 2.4a zeigt ein zu Abbildung 2.4 vergleichbares Publisher/Subscriber System aus 3 Knoten.

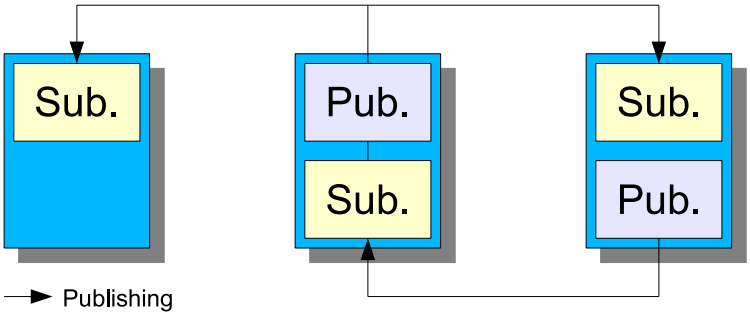


Abbildung 2.4a: Publisher/Subscriber-Modell

Ein weiterer Unterschied zum Client/Server-Modell ist die Initialisierung von Sendeoperationen. Im Client/Server-Modell werden Daten ausschließlich auf explizite Anforderung des Clients versendet. Es ist somit ein reines Pull-System. Aktivitäten können immer nur vom Client gestartet werden. Ein Server kann einen Client zum Beispiel nicht über wichtige Änderungen der Daten informieren. Beim Publisher/Subscriber-Modell ist die Initialisierung der Übertragung genau anders herum. Publisher senden Daten unabhängig von der Existenz von Subscribern für diesen Inhalt. Auslöser für das Senden der Publisher können I/O Ereignisse oder auch zeitliche Ereignisse sein. Das Publisher/Subscriber-Modell ist somit eine Push-System.

(vgl. [EUGSTER])

2.5 Ereignisbasierte Systeme

Ein **Ereignis** ist die „Spezifikation eines wichtigen Vorkommnisses, das an einem bestimmten Zeitpunkt an einem bestimmten Ort eintritt. Im Kontext eines Automaten ist ein Ereignis das Eintreffen eines Auslösers, der eine Zustandsänderung auslösen kann. Im Kontext eines Aktivitätsdiagramms kann ein Ereignis eine Aktivität auslösen“ ([UML2]).

Systeme, deren Programmablauf von Ereignissen gesteuert wird, nennt man **Ereignisbasierte Systeme**. Ereignisse werden von externen Komponenten eines Gesamtsystems initialisiert. Das System kann auf diese Ereignisse mit entsprechenden Aktionen reagieren. Unter hinreichend generalisierter Sicht ist jedes System ereignisbasiert. Selbst eine simple Addition benötigt zumindest die Initialisierung als fundamentales Ereignis. Hier werden Systeme aber nur ereignisbasiert genannt, wenn deren Programmablauf auf die Reaktion zu Ereignissen ausgelegt ist. Die Funktion ereignisbasierter Systeme ist damit von Umwelteinflüssen abhängig. Diese befinden sich im Allgemeinen außerhalb des Einflussbereichs des Systems.

Ereignisse können vom System gepollt werden. Das heißt, dass das System ständig überprüft, ob ein Ereignis eingetreten ist. Da es dann aber durchgängig mit Abfragen des Ereignisstatus beschäftigt ist, wird die Rechenleistung selbst im Ruhezustand, also wenn kein Ereignis auftritt, vollständig ausgelastet. Diese Situation ist für Multitaskingumgebungen nicht akzeptabel, denn hier ist Rechenleistung eine gemeinsame Ressource. Um die Last zu vermeiden, kann mit Zeitslots gearbeitet werden. Die ständige Last ist dann nicht mehr gegeben. Nachteil ist aber, dass das System nur noch in vorgegebenen Perioden, nämlich genau den Zeitslots, auf Ereignisse reagieren kann. Als Folge dessen ist die Reaktionsfähigkeit drastisch eingeschränkt. Eine bessere Lösung bieten hier Unterbrechungen. Durch Ereignisse werden entsprechende Funktionen der Zielkomponenten aufgerufen. Die Reaktionsfähigkeit bleibt vollständig erhalten, und das System muss nur bei Ereignissen arbeiten. Im Ruhezustand wird auch keine Rechenleistung benötigt (vgl. [GEA]).

In verteilten Systemen können Netzwerkkomponenten, wie der Treiber einer Netzwerkkarte, Quelle dieser Ereignisse sein. Dort ist der Empfang eines Datenpakets ein Ereignis, das eine bestimmte Zielkomponente zur Verarbeitung des Paketes aktiviert. Weitere, besonders für Echtzeitsysteme relevante Ereignisse, sind zeitliche Ereignisse. Häufige Anwendungen dieser Ereignisse sind Time-outs, um Deadlines für Echtzeitoperationen zu implementieren.

2.6 GEA

GEA ist eine Programmierschnittstelle, die alle notwendigen Funktionen zur

Entwicklung von ereignisbasierten Netzwerkprotokollen bereitstellt. Sie wurde an der Otto-von-Guericke-Universität Magdeburg von der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ an der Fakultät für Informatik entwickelt. Die Motivation zur Entwicklung von GEA lag in der Vereinfachung des Entwicklungsprozesses ereignisbasierter Netzwerkprotokolle. Typische Schritte der Entwicklung sind hier die Implementierung auf einer Simulationsplattform, gefolgt von Implementierungen auf Plattformen, auf denen das Protokoll zum Einsatz kommen soll. Simulations- und Einsatzplattformen haben im Allgemeinen unterschiedliche Schnittstellen zu Primitiven, die für die Implementierung des Protokolls notwendig sind. Je Plattform ist dann im schlimmsten Fall eine Implementierung notwendig. Da diese Implementierungen dann auch parallel gewartet werden müssen, sie aber auch gleichzeit unterschiedliche Fehler und Implementierungsprobleme hervorrufen, wird die Aufmerksamkeit der Entwickler mehr auf diese Probleme gezogen, anstatt auf die Algorithmen des Protokolls. Eine effizientere Entwicklung, die damit dann auch ein qualitativ höherwertiges Ergebnis zur Folge hat, lässt sich erzielen, wenn es nur eine Implementierung für alle Simulations- und Einsatzplattformen gibt. Um dies zu realisieren, wird GEA als Abstraktionsschicht zwischen Plattform und Protokoll benutzt.

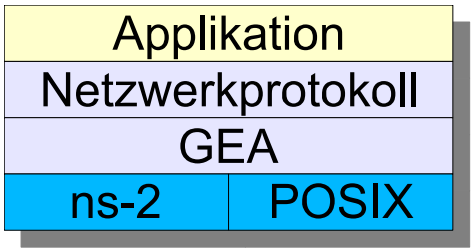


Abbildung 2.6: Einordnung GEA

Auf allen Plattformen werden dann von GEA dieselben Primitiven zur Verfügung gestellt. Netzwerkprotokolle können dann mit ein und derselben Implementierung auf allen Plattformen betrieben werden, für die eine Portierung von GEA existiert. Der Aufwand für die Pflege verschiedener Implementierungen ist damit nicht mehr gegeben. Die Portierungsprobleme bei der Entwicklung sind auf GEA konsolidiert. Um die Portierung von GEA auf unterschiedliche Plattformen so unkompliziert wie möglich zu gestalten, wurde das Minimum an benötigten Primitiven ermittelt, die für die Entwicklung von ereignisbasierten Netzwerkprotokollen nötig sind. Dies sind die folgenden vier Funktionen:

- Senden von Daten
- Empfangen von Daten
- Warten für eine bestimmte Zeit
- paralleles Warten auf I/O Aktivitäten mit Timeout

(vgl. [GEA])

Zum Zeitpunkt dieser Diplomarbeit ist GEA auf zwei Plattformen portiert. Zum einen wird der Netzwerksimulator **ns-2** unterstützt. In Kapitel 2.7 wird genauer auf diesen Simulator eingegangen. Die zweite Plattform ist POSIX. **POSIX** ist die Abkürzung für „Portable Operation System Interface“, das von „The Open Group“ gepflegt wird.

Beim IEEE ist es als Standard 1003 geführt. Es definiert eine standardisierte Betriebssystemschnittstelle, die auf Ebene des Quellcodes die Portabilität von Applikationen zwischen verschiedenen Betriebssystemen ermöglicht (vgl. [POSIX]). Von den meisten UNIX Derivaten wird der POSIX Standard größtenteils erfüllt. Andere Betriebssysteme, wie Microsoft WindowsNT und folgende, sind auch teilweise konform zu diesem Standard. Entwickelt und getestet wurde die POSIX Portierung von GEA auf Linux, da es die primäre Entwicklungsumgebung der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ ist. Linux ist OpenSource, wodurch keine eingeschränkte Verfügbarkeit existiert. Dies wiederum erhöht die Chance auf breite Akzeptanz von GEA.

Neben der Bereitstellung der Kommunikationsschnittstelle bietet GEA die Funktion als globaler Event-Handler. Anwendungen (Protokolle) registrieren für verschiedene Ereignisse eigene Handler in Form von Callbackfunktionen. Neben I/O-Ereignissen, die zum Beispiel durch den Empfang eines Paketes hervorgerufen werden, managt er auch zeitbasierte Ereignisse. Dafür wird ein einheitliches Zeitmodell bereitgestellt. Über dieses Zeitmodell ist es möglich, echtzeitfähige Netzwerkprotokolle zu implementieren. Durch die Behandlung der zwei Ereignisarten mit einheitlichen Methoden vereinfacht GEA die Entwicklung von ereignisbasierten Protokollen enorm. Der Entwickler kann sich voll auf die Funktion des Protokolls konzentrieren und muss sich nicht mit dem Overhead plattformspezifischer Lösungen beschäftigen.

In Abbildung 2.6 wird deutlich, dass nur auf Basis von GEA entwickelt wird. Um die gewünschte Portabilität zu gewährleisten, dürfen keine Primitive der Plattform direkt genutzt werden. Allerdings entsteht bei Abstraktionsschichten die Gefahr, dass durch die zusätzliche Zwischenverarbeitung von grundlegenden Protokollfunktionen die Performance des resultierenden Protokolls beinträchtigt wird. Minimale Verzögerungen gegenüber einem direkten Zugriff können sich durch gehäufte Nutzung der entsprechenden Primitive zu großen Gesamtverzögerungen summieren. Da GEA vor allem auf die Entwicklung echtzeitfähiger Protokolle zielt, wurde hier besonders darauf geachtet. Die Funktionen von GEA sind auf das absolute Minimum für ereignisbasierte Protokolle beschränkt. Diese sind bereits zuvor aufgelistet. Wichtig für diese Funktionen ist aber zusätzlich, dass ihre Implementierung mit geringer Komplexität realisiert sein muss.

Bei dem zu entwickelnden Cachesystem handelt es sich nicht um ein eigenständiges Netzwerkprotokoll. Vielmehr ist es eine Komponente für die Publisher/Suscriber-Middleware, die zur Zeit von der Arbeitsgruppe entwickelt wird. Im nächsten Kapitel wird diese kurz beschrieben. Jegliche Netzwerkkommunikation wird somit nicht direkt über GEA, sondern über die Middleware realisiert. Für zeitliche Ereignisse wird GEA aber direkt benutzt. Zu beachten ist dabei, dass die Handlerfunktionen für eintretende Ereignisse mit möglichst geringem Zeitaufwand ausgeführt werden müssen. Andernfalls wird die Reaktionsfähigkeit der gesamten Middleware in Mitleidenschaft gezogen, da diese ebenfalls GEA nutzt, und somit ein einziger Event-Handler verwendet wird. Da kein eigenes Zeitmodell mit zugehörigem Event Handler entwickelt werden muss, ist der Implementierungsaufwand stark reduziert. Ergebnisse der Entwicklungen sind schneller verfügbar. Zusätzlich wird durch die reduzierte Komplexität auch die Wahrscheinlichkeit für Programmierfehler geringer, was ein qualitativ höherwertiges System zur Folge hat.

In der ersten Phase der Entwicklung des Cachesystems wird Linux als Plattform genutzt. Hier können Basisfunktionen wie die Speicherstruktur und die grundlegende Kommunikation durch Standardausgaben ideal getestet werden. In der zweiten Phase wird auf den ns-2 gewechselt. Wie schon beschrieben, ist das ohne Änderungen des Quellcodes möglich. Auf dem ns-2 kann die Zusammenarbeit der Cacheknoten unter realitätsnahen Bedingungen simuliert und getestet werden. Jetzt werden die entsprechenden Funktionen für die Zusammenarbeit realisiert und überprüft. In der dritten Phase wird wieder auf Linux gewechselt. Analog zur vorherigen Phase ist keine Codeanpassung nötig. Nun kann das entstandene System in realen Umgebungen getestet werden.

2.7 Publisher/Subscriber Middleware

Das zu entwickelnde Cachesystem wird im Projekt „Eine Publisher/Subscriber-basierte Middleware mit Dienstgütegarantien zur Unterstützung kooperativer Anwendungen“ integriert. Ziel des Projektes ist es, eine Umgebung für Multi-Hop-Funknetzwerke bereitzustellen, die echtzeitfähige Kommunikationen ermöglicht. Dabei soll aber kein neues Netzwerk entstehen. Vielmehr soll die Middleware auf einem Netzwerk arbeiten, das keine Dienstgütegarantien selbst bereitstellt. Ziel ist die Integration in IEEE802.11 Netzwerke (WLAN). Die Middleware verwaltet dafür die Ressourcen des Netzes und sorgt mit Lastkontrolle und -verteilung für die Einhaltung der Dienstgüten, die von den Applikationen gefordert werden.

Die Kommunikation auf Basis des Client/Server-Modells ist in Ad-Hoc-Netzwerken problematisch. Verbindungen zwischen zwei Endpunkten können auf Grund der inhärent hohen Dynamik solcher Netze leicht abbrechen oder nur über hohe Kosten aufrechterhalten werden. Des Weiteren sind Daten im Anwendungsfeld dieser Netze oftmals lokalitätsgebunden. Das Publisher/Subscriber-Modell (siehe Kapitel 2.4) ermöglicht hier inhaltsbasierte Kommunikation. Es gibt somit keine direkte knotenbasierte Adressierung. Die resultierenden Kommunikationsverbindungen sind im Gegensatz zum Client/Server-Modell nicht 1-zu-1, sondern n-zu-m. Der Verlust einzelner Knoten ist in diesem Zusammenhang nicht relevant für die Aufrechterhaltung der Kommunikation. Die Verfügbarkeit von Inhalten ist von einzelnen Knoten unabhängig.

(vgl. [PSMW], [PSMWDFG])

Die PSMW (Publisher/Subscriber-Middleware) clustert das Netzwerk wie in Kapitel 2.3 beschrieben. Um Dienstgütegarantien in der Übertragung von Daten zu ermöglichen, wird die Kommunikation innerhalb eines Clusters mit einem TDMA (Time Division Multiple Access) Protokoll und PCF (Point Coordination Function) durchgeführt. Durch das TDMA werden Zeitslots zur Kommunikation verwendet. Einem Knoten wird dabei immer ein Zeitslot zugewiesen, in dem er das Medium zur Verfügung hat und eventuell anstehende Daten senden kann. Um die Zuweisung der Zeitslots deterministisch zu gestalten, was Voraussetzung für echtzeitfähige Kommunikation ist, wird PCF verwendet. Die Zeitslots werden dabei zentral von einem Koordinator vergeben (vgl. [802.11] 86ff). Diese Aufgabe übernimmt der

Clusterhead. Die Zuweisung der Slots alteriert zwischen Clusterhead und Clients. Nach dem Versenden anstehender Daten weist der Head den nächsten Slot genau einem Client zu. Alle Clients bekommen so im Round Robin Verfahren die Slots gleichmäßig zugewiesen. Hatten alle Clients einmal einen Slot zugewiesen und konnten Daten versenden, wird dies als Round-Trip bezeichnet. Danach beginnt der Clusterhead die Slotzuweisung wieder von vorn. Ein Round-Trip dauert genau zweimal so viel Slots, wie der Cluster Clients hat. In Abbildung 2.7 ist ein Cluster mit vier Mitgliedern mit der zugehörigen Slotzuweisung über zwei Round-Trips dargestellt. Ein Round-Trip dauert hier sechs Slots.

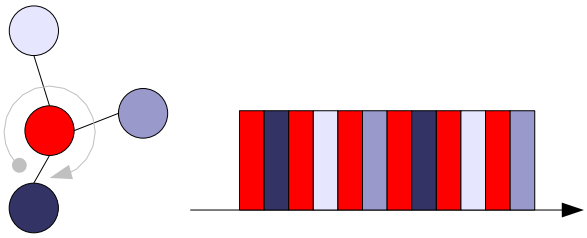


Abbildung 2.7: TDMA mit PCF

Das Ergebnis ist eine deterministische Verteilung der verfügbaren Bandbreite. Bei bekannter Mitgliederanzahl im Cluster ist die Bandbreite, die einem Mitglied zugesichert wird, auf einfachem Wege berechenbar. Das Scheduling von Echtzeitübertragungen ist innerhalb eines Clusters damit realisierbar.

Neben Echtzeitübertragungen unterstützt PSMW auch Best-Effort-Übertragungen. Daten, die mit Best-Effort übertragen werden, werden immer hinter Echtzeitdaten zurückgestellt. Best-Effort-Daten werden somit immer nur dann übertragen, wenn ein Slot mit Echtzeitdaten nicht ausgelastet ist. Da durch das PCF die Slots unabhängig von anstehenden Übertragungen zugewiesen werden, stellt die Ausnutzung ungenutzter Bandbreite durch Best-Effort-Daten keinerlei Beeinträchtigung der Echtzeitkommunikation dar. Für die Übertragung von Best-Effort-Daten stehen drei Kommunikationsprimitiven zur Verfügung. Zum einen gibt es **CellBroadcast**. Pakete werden dabei zum Clusterhead geschickt, der sie anschließend an alle Clients verschickt. Der zeitliche Aufwand für einen CellBroadcast beträgt minimal zwei Slots. Eine Aussage über die maximale Dauer eines CellBroadcasts lässt sich wegen der Unterordnung gegenüber Echtzeitdaten nicht machen. Es ist die einfachste Kommunikation, die in einem PSMW-Cluster möglich ist. Es gibt allerdings keine Rückmeldungen über den Erfolg der Übertragung. Da auch eine variierende Fehlerrate besonders in Funknetzwerken immer vorauszusetzen ist, muss immer davon ausgegangen werden, dass Pakete nicht alle Clients erreichen. Kommunikation, die auf CellBroadcast basiert, kann somit keinerlei Zusicherung über den Zustellerfolg von Paketen haben. Eine Verbesserung ist durch die Einführung von Empfangsrückmeldungen zu erwarten. Erreichen diese nicht vollständig für alle Clients den Head, muss er von einer erfolglosen Übertragung ausgehen. Das Paket wird dann vom Clusterhead erneut versendet, bis der Clusterhead die Rückmeldungen aller Clients empfängt. Wird innerhalb einer bestimmten Anzahl von Wiederholungen der Status einer erfolgreichen Übertragung nicht erreicht, wird sie endgültig abgebrochen. Dies ist erforderlich, um keine Endlosversuche zu verursachen. Der initiierenden Applikation kann dann eine Statusrückmeldung gegeben werden. Diese Vorgehensweise nennt sich **Reliable CellBroadcast**. Sie

liefert mit hoher Wahrscheinlichkeit eine vollständige Übertragung, bzw. eine konkrete Rückmeldung über den Erfolg. Nachteil ist allerdings, dass eine Übertragung erst nach mindestens einem Round-Trip als erfolgreich gelten kann. Dies ist auch der Fall, wenn alle Clients auf Anhieb das Paket empfangen. Damit jeder Client Rückmeldung geben kann, muss der Clusterhead auch erst jeden Client mindestens einmal einen Slot zugewiesen haben. Kann nur ein Client keine Rückmeldung senden, weil sein Slot mit Echtzeitdaten ausgelastet ist, reicht ein Round-Trip schon nicht mehr aus. In realen Netzen mit einer annehmbaren Fehlerrate und einer moderaten Auslastung mit Echtzeitverkehr kann es dann wahrscheinlich sein, dass mehrere Versuche benötigt werden, was zur Folge hat, dass auch mehrere Round Trips benötigt werden. Der Zeit- und Bandbreitenaufwand ist gegenüber dem einfachen CellBroadcast immens.

Nun ist das Netz mit den beiden Übertragungsvarianten noch nicht in der Lage, über Clustergrenzen hinweg Best-Effort-Daten zu senden. Hier kommt der globale **Broadcast** ins Spiel. Dieser entspricht der Flutung des gesamten Netzes. Die Basis ist der CellBroadcast. Rückmeldungen über den Erfolg einer Übertragung sind somit nicht möglich. Bei der Übertragung eines Broadcast haben die Gateways eine zentrale Rolle. Sie leiten ein Broadcastpaket des einen Clusters an die Clusterheads des oder der anderen Cluster weiter. Diese versenden ihn wiederum an ihre Clients. Das Broadcastpaket verbreitet sich kaskadierend durch das gesamte Netz. Durch Zyklen in der Netzwerktopologie können diese Pakete Cluster mehrmals erreichen. Um dann Schleifen in der Verbreitung zu verhindern, werden Sequenznummern genutzt. Ein globales Äquivalent zum Reliable Broadcast ist nicht vorgesehen. Der Aufwand für Rückmeldungen würde bei der theoretisch unbegrenzten Anzahl möglicher Knoten eines Netzes mit mehreren Clustern unberechenbare Dimensionen annehmen. Eine solche Methode ist als nicht sinnvoll anzusehen.

2.8 Netzwerksimulator ns-2

Der Netzwerksimulator **ns-2** wird am „Information Sciences Institute“ der „University of Southern California“ entwickelt. Er ist ein diskreter, ereignisbasierter Netzwerksimulator. Simulationen werden dabei über zeitlich gesteuerte Ereignisse beschrieben, die zu diskreten Zeitpunkten aktiviert werden. Die Ereignisqueue des Simulators arbeitet zeitlich unabhängig von Reaktionen und zeitlichen Schemata der simulierten Systeme. Die Arbeitsweise der Ereignisqueue entspricht der von GEA. Für die Simulation von GEA-basierten Netzwerkprotokollen ist der ns-2 dadurch ideal geeignet. Im Umkehrschluss war die Portierung von GEA auf den ns-2 in Bezug auf die Verarbeitung zeitlicher Ereignisse mit direkter Abbildung realisierbar. Simulationsergebnisse sind demnach als sehr hochwertig anzunehmen, weil nahezu kein Overhead durch komplexe Schnittstellenanpassungen nötig ist.

Die Struktur ist modular aufgebaut. Netzwerkmodelle repräsentieren verschiedene Netzwerktypen mit den entsprechenden Eigenschaften. Hierunter fallen drahtgebundene Netze und drahtlose Netze, wie IEEE802.11[a|b|g]. Besonders relevant für die Arbeit ist hier die Unterstützung von Ad-Hoc-Netzwerken. Viele Routingprotokolle sind fertig implementiert verfügbar. Neben puren IP-Übertragungen

können Unicast UDP- und TCP-Verbindungen sowie Multicast-Verbindungen für eigene Entwicklungen verwendet werden. Um Protokolle zu testen, können übliche Datenquellen wie HTTP, FTP und Telnet eingesetzt werden. Mit Zufallsdatenquellen hat man die Möglichkeit, Fehler im Netzwerk zu initiieren. Für die Entwicklung von Protokollen können implementierte Queuemethoden verwendet werden.

Szenarien werden im ns-2 mit OTcl beschrieben. OTcl ist eine objektorientierte Erweiterung von Tcl/Tk (vgl. [OTCL]). Als Ergebnis liefert der ns-2 einen Trace, der normalerweise auf der Standardausgabe ausgegeben wird. Der Trace lässt sich auch umleiten, um diesen weiter zu verwenden. Ziel dieser Umleitung könnte **nam** sein. nam ist der „Network Animator“ von ns-2. Er ermöglicht die grafische Auswertung der Simulationen. Dabei wird der Zustand des simulierten Netzes und die Übertragungen, über die Zeit betrachtet, animiert dargestellt.

Der ns-2 ist OpenSource, wodurch er einem ständigen Entwicklungsprozess unterliegt. Jeder Nutzer, der eine Erweiterung des Simulators benötigt, kann diese selbst entwickeln. Der Simulator kann auf den Plattformen FreeBSD, Linux, Solaris, Windows und Mac betrieben werden.

(vgl. [NSTUT], [NSWL])

3 Konzept

Als Komponente der Publisher/Subscriber Middleware (PSMW, siehe Kapitel 2.7) ist der Cache primär zuständig für die verteilte Speicherung von systemnahen Informationen. Dazu gehören Topologieinformationen, die für das Routing in der PSMW von elementarer Bedeutung sind, sowie Inhaltsbeschreibungen der Publisher und der Subscriber. Letztere ermöglichen die Adressierung der Multicast-Übertragungen, die für das Publisher/Subscriber-Modell (siehe Kapitel 2.4) notwendig sind. Abbildung 3 zeigt die strukturelle Einbettung des Cachesystems in der PSMW.

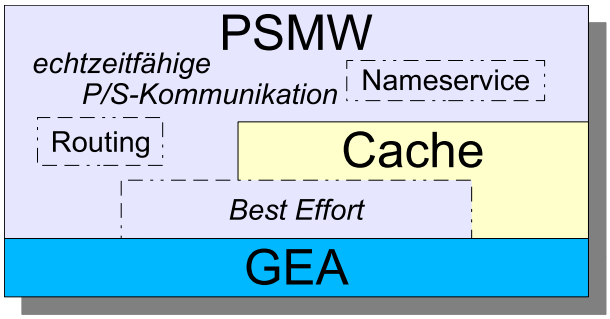


Abbildung 3: strukturelle Einbettung

Das Cachesystem soll keine echtzeitfähige Publisher/Subscriber Kommunikation benutzen, da es als Beiwerk zur PSMW die verfügbare Bandbreite für Echtzeitkommunikation nicht beeinträchtigen soll. Infolgedessen steht für die Entwicklung des Cachesystems nur Best Effort Kommunikation zur Verfügung. Best-Effort-Übertragungen werden von der PSMW immer hinter Echtzeitübertragungen zurückgestellt. Die möglichen Varianten der Best-Effort-Kommunikation wurden in Kapitel 2.7 besprochen. Reliable CellBroadcast ist dabei die einzigste Möglichkeit Zustellungen zu kontrollieren. Allerdings werden sehr häufig Cachelines übertragen. Der Aufwand für Reliable CellBroadcast ist dafür zu hoch. Unter hoher Auslastung des Netzes durch Echtzeitkommunikationen ist die verfügbare Bandbreite für Best-Effort Daten sehr gering. Einzelne Knoten haben eventuell nicht die Möglichkeit, die nötige Rückmeldung zu senden. Die Anzahl der Round-Trips, die für den Reliable CellBroadcast benötigt werden, kann dadurch sehr hoch werden. Der Übertragungsaufwand muss reduziert werden. Als Alternative bleibt nur der normale CellBroadcast. Die Anzahl an Cachelines, die verglichen zum Reliable CellBroadcast innerhalb eines identischen Zeitraumes übertragen werden können, ist weitaus höher, da sie einfach nacheinander versendet werden und keine Rückmeldungen der Empfänger nötig sind. Allerdings müssen dadurch mögliche Übertragungesverluste beachtet werden. Bis hierhin kann innerhalb eines Clusters kommuniziert werden. Für die Verbreitung von Cachelines über Clustergrenzen hinweg kann laut Kapitel 2.7 nur der globale Broadcast, also die Netzwerkflutung, verwendet werden. Es ist die einzige Möglichkeit der PSMW Best-Effort-Pakete clusterübergreifend zu versenden. Allerdings muss, um eine Verbesserung gegenüber der benötigten Bandbreite ohne Einsatz des Cachesystems zu erreichen, auf die Flutung verzichtet werden. Die Lösung ist hier eine Diffusion der Daten von Cluster zu Cluster. Durch Optimierung dieses Diffusionsprozesses ist eine deutliche Lastverbesserung gegenüber vorher zu

erwarten. Die einzige einsetzbare Datenübertragung bleibt der CellBroadcast. Da dieser keine Kontrolle über die Zustellung von Paketen hat, kann das Cachesystem nicht mit 100%iger Datenkonsistenz arbeiten. Deshalb wird in Kapitel 3.7 eine stochastische Analyse des Systems durchgeführt, nachdem die Struktur vollständig entwickelt wurde und auf Details eingegangen werden kann.

Die Entwicklung des Cachesystems wird von zwei konkurrierenden Anforderungen begleitet. Zum einen muss das System möglichst ressourcenschonend, wünschenswert ist sogar unauffällig, arbeiten. Unauffällig ist hier so zu verstehen, dass die Bandbreite für Nutzübertragungen in der PSMW nicht beeinträchtigt wird. Dies ist bereits durch die Beschränkung auf Best-Effort-Übertragung per CellBroadcast gegeben. Zum anderen wird maximale Zuverlässigkeit erwartet, damit die Komponenten der PSMW sich auf die Verarbeitung der Cachelines verlassen können. Im Idealfall soll es eine Plattform für globales Wissen darstellen. Unter den Bedingungen des Netzes kann allerdings nur ein System entstehen, das in allen Bereichen hochgradig tolerant mit Fehlern umgeht. Lücken in der Datenübertragung dürfen nicht zum Kollaps des Systems führen.

In Client/Server-Systemen werden Caches eingesetzt, um die Last auf Server durch Zugriffe großer Mengen von Clients zu reduzieren. Clients stellen ihre Anfragen dann an die Caches anstatt direkt an den Server. Die Cachesysteme speichern die Antworten auf Clientanfragen zwischen, um die Anfragen selbst zu beantworten. Ist die lokale Beantwortung nicht möglich, leiten Caches die Anfrage stellvertretend für die Clients an den Server weiter. Ein zweiter Grund für den Einsatz von Caches ist die Reduktion von Netzwerkverkehr. Werden solche Cachesysteme an Infrastrukturknotenpunkten eingesetzt, kann Datenverkehr dort abgefangen werden, der sonst gebündelt weitergeleitet würde. Hier ist die Motivation für die Nutzung eines Cachesystem ebenfalls die Reduktion des Netzwerkverkehrs. Allerdings gibt es keine dedizierten Cacheknoten. Alle Knoten des Netzes sind Teil des Caches. Jeder Knoten ist in der Lage, eine Anfrage an den Cache lokal auszuführen. Ein sehr relevanter Unterschied zum klassischen Cache in Client/Server-Umgebungen ist, dass hier, bei negativem Ergebnis einer Suche im Cache, keine Anfrage, stellvertretend für die anfragende Applikation, an den Informationsursprung gestellt wird. Diese Aufgabe obliegt der Applikation selbst. Im Fall des Namensdienstes der PSMW, der Informationen darüber zur Verfügung stellt, welcher Knoten Subscriber für welche Inhalte ist, muss dieser dann nach negativer Suche einer Liste von Subscribern für einen bestimmten Inhalt die Informationen über eine Flutung ermitteln. Das Ergebnis der Flutung wird dann zur späteren Wiederverwendung in das Cachesystem eingetragen.

3.1 Verteilung

Um Cachesysteme vor zu hoher Last zu bewahren, und den Umfang der benötigten Ressourcen der Systeme gering zu halten, werden sie verteilt eingerichtet. Die zwischenzuspeichernden Daten können so nach diversen Methoden auf mehrere Knoten des Caches verteilt werden. Im Kontext dieser Arbeit steht die Reduktion der Netzwerklast und der Antwortzeiten sowie die Minimierung der benötigten

Ressourcen im Mittelpunkt. Im Hinblick auf diese Ziele betrachte ich im Folgenden Verteilungsmöglichkeiten auf ihre Tauglichkeit für den Einsatz in der PSMW.

Die Clusterstruktur aus Kapitel 2.2 bildet die Netzwerkstruktur, in der sich das System befindet. Da ein Cluster dort eine Kommunikationseinheit darstellt, bietet es sich an, diesen auch als eine Einheit des Cachesystem zu verstehen. Die billigste Kommunikationsform, der CellBroadcast, deckt damit alle Mitglieder einer Cacheeinheit ab. Teure Übertragungen über die Clustergrenzen hinweg sind somit nicht nötig. Im Folgenden werden diese Einheiten nur noch als Cluster bezeichnet. Sie arbeiten unabhängig voneinander. Dadurch wird die Komplexität des Cachesystem gering gehalten, denn außer der des CellBroadcast sind keine anderen Kommunikationsformen nötig.

Durch diese strikte Abgrenzung hat der Cache nur Daten aus dem eigenen Cluster und Daten anderer Cluster, die von den eigenen Clustermitgliedern ermittelt wurden. Für den Nutzen des Cachesystems ist dies nicht akzeptabel. Eine proaktive Bereitstellung von Informationen über das gesamte Netzwerk, wie es für das Routing und den Namensdienst hilfreich und notwendig ist, ist so nicht realisierbar. Auf Datenaustausch zwischen den Clustern kann nicht verzichtet werden. Die Verwendung des globalen Broadcast ist dafür aber nicht geeignet. Bei jeder Änderung der Daten eines Clusters wird sonst ein Broadcast ausgelöst. Der durch diesen Broadcast hervorgerufene Datenverkehr ist im Vergleich zur Brisanz der zu übertragenen Information im Allgemeinen nicht angemessen. Denn allein der erfolgreiche Zugriff auf eine Cacheline, also ein **Hit** in der Terminologie von Cachesystemen, hat dann einen Broadcast zur Folge. Deshalb wird die Verbreitung von Cachelines in andere Cluster als eine Art Datendiffusion realisiert. Dabei werden die Informationen verzögert und gebündelt verbreitet.

In den folgenden Unterkapiteln wird die detaillierte Entwicklung des eben kurz beschriebenen Systems erläutert. Die Diffusion der Daten wird in Kapitel 3.5 beschrieben. Bis dahin liegt der Fokus auf der Arbeitsweise innerhalb eines Clusters.

3.1.1 ROWA

In einem statischen Infrastrukturnetzwerk würde der Cache auf einem dedizierten Knoten zentral arbeiten. Übertragen auf die Clusterstruktur könnte diese Aufgabe der Clusterhead übernehmen, da jeglicher Datenverkehr über ihn geleitet wird. Da es sich aber um Ad-Hoc-Netze handelt, die per Definition hochgradig dynamisch sind, können aber jederzeit Knoten durch Verbindungsverluste oder Knotenausfälle aus einem Cluster austreten. Betrifft dies den Master, übernimmt zwar ein anderer Knoten seine Aufgaben, allerdings sind sämtlich Cachedaten verloren, so dass wieder mit einem leeren Cache angefangen wird. Ein erster Ansatz, um dieses Problem zu lösen, ergibt sich, indem jeder Knoten mit einem eigenen Cache ausgestattet ist. Die Cachesysteme sind dann datenunabhängig. Allerdings kann man in dieser einfachen Ausführung nicht von einem verteilten Cachesystem sprechen. Ohne Beziehungen zwischen den Caches handelt es sich nur um Clientcaching (vgl. [Bengel] 61ff). Um dem Anspruch eines verteilten Caches gerecht zu werden, können alle Caches eines Cluster konsistent aktualisiert werden. Es

entsteht ein **ROWA** (Read Once / Write All) System. Da es sich lediglich um einen Cache, und nicht zum Beispiel ein Datenbankmanagementsystem handelt, sind die Konsistenzanforderungen nicht sehr hoch. Kleine Unterschiede in den Ständen der Daten sind ohne weiteres tolerierbar. Updateprotokolle, wie das **2 Phase Commit Protokoll**, die konsistente Updates garantieren, sind somit nicht notwendig. Jeder Knoten müsste das Update bestätigen, anderenfalls gilt es als nicht durchgeführt. Dieses Vorgehen entspricht dem Reliable Broadcast. Es arbeitet nur eine Ebene darüber. Die zu speichernden Daten in einem Cache sind generell temporär. Sie altern und verlieren darüber ihre Gültigkeit. Da der Anspruch 100%iger Konsistenz hier nicht gegeben ist, fällt auch das Hauptproblem des ROWA-Ansatzes in Umgebungen mit Kosistenzanforderung weg. Ist nur ein Knoten nicht in der Lage, die Daten zu empfangen oder eine Bestätigung zu senden, ist das gesamte System nicht in der Lage, Änderungen durchzuführen. Es ensteht ein fehlertolerantes Replikationssystem.

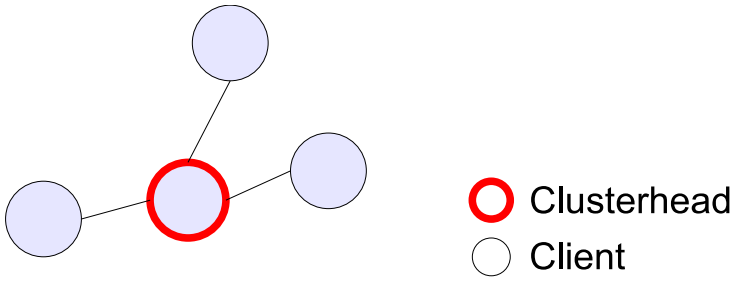


Abbildung 3.1.1: ROWA-Cache

Abbildung 3.1.1 zeigt einen Cluster, in dem die Verteilung nach dem ROWA-Prinzip realisiert ist. Alle vier Knoten haben mit einer gewissen Toleranz an Differenzen den selben Datenbestand. Die Differenzen entstehen durch die zuvor beschriebene Datenübertragung. Das System ist nicht in der Lage, Übertragungen zu garantieren. In den folgenen Betrachtungen wird der Einfachheit halber eine konsistente Kommunikation angenommen. Die ROWA-Methode ist leicht zu implementieren. Jedes Update wird per CellBroadcast propagiert. Da es keine disjunkte Verteilung der Daten gibt, verarbeitet jeder Knoten jede propagierte Änderung. Neben der einfachen Implementierung ergibt sich auch der Vorteil maximaler Redundanz. Selbst wenn alle Knoten, außer einem, den Cluster verlassen, entsteht kein Datenverlust. Gerade in Bezug auf den Einsatz in eingebetteten Systemen ergibt sich daraus aber auch der Hauptnachteil. Die starke Redundanz widerspricht der Forderung nach sparsamem Umgang mit den Ressourcen. Je größer ein Netz wird, um so mehr Daten müssen gespeichert werden. Da alle Knoten eines Clusters den selben Datenbestand haben, steigt der Ressourcenbedarf linear mit.

3.1.2 Hashverteilung

Im vorherigen Kapitel wurde deutlich gemacht, dass es einer disjunkten Verteilung der Daten auf mehrere Knoten bedarf. Dazu müssen alle vorhandenen Daten auf einen begrenzten diskreten Wertebereich abgebildet werden. Eine weitere Anforderung an diese Abbildung ist Eindeutigkeit. Cachelines, die anhand dieser Abbildung auf mehreren Knoten verteilt wurden, müssen über diese Abbildung auch

wieder gefunden werden. Basis dafür ist die Behandlung von Cachelines als Schlüssel/Wert-Paar. Die Abbildung bezieht sich dabei immer auf den Schlüssel. Bei einer Suche nach einem Wert zu einem bestimmten Schlüssel wird die Abbildung dann genutzt, um den korrekten Teil, in dem die gesuchte Cacheline sich befindet, zu ermitteln. Hashfunktionen erfüllen die Anforderungen. Eine **Hashfunktion** bildet eine offene Menge von Werten auf eine geschlossene Menge von Werten eindeutig ab. Das intuitivste Beispiel einer Hashfunktion stellt die Modulooperation dar. Sie bildet ganze Zahlen auf Restklassen ab.

$$\begin{array}{rcll} h(k) & = & k \bmod m & h(k) \in \{0,\ldots,m-1\} \\ k & : & \textit{zu hashender Wert} & k \in \mathbb{Z} \\ m & : & \textit{Anzahl der Restklassen} & m \in \mathbb{N} \end{array}$$

Im Kontext der Verteilung stellt m die Anzahl disjunkter Cacheteile dar. k ist der Schlüssel, dessen Wert gespeichert oder abgefragt werden soll. Je nach Ergebnis der Funktion $h(k)$ wird das Schlüssel/Wert-Paar in einem der m Restklassen gespeichert oder von dort abgefragt.

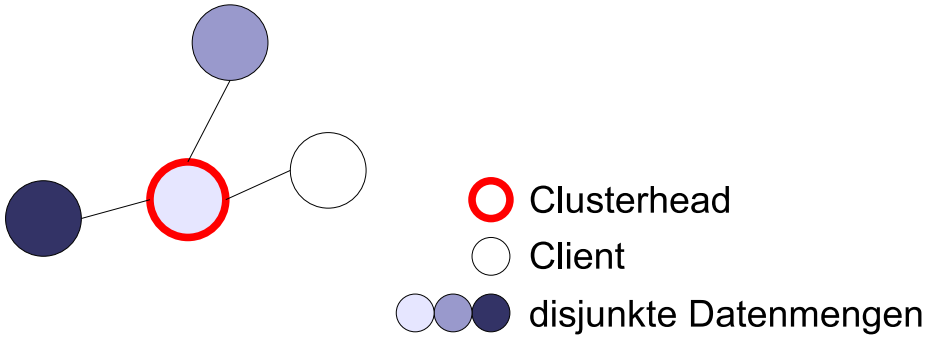


Abbildung 3.1.2: einfache Hashverteilung

In der Abbildung 3.1.2 ist der Cluster aus Abbildung 3.1.1 mit einer Hashverteilung dargestellt. Die verwendete Hashfunktion ist eine Modulooperation mit drei Restklassen. Der Cache wird somit disjunkt auf drei Knoten des Clusters verteilt. Die Information, welcher Knoten welche Restklasse hält, muss auf jedem Knoten konsistent vorhanden sein, um effektiv mit dem Cache arbeiten zu können. In dem Beispiel ist absichtlich nur eine Menge von drei Restklassen gewählt, obwohl vier Knoten vorhanden sind, um ein grundlegendes Problem dieses Ansatzes zu verdeutlichen. Die Aufteilung auf mehrere Knoten ist hier nur statisch. Da ein Cluster aber dynamisch ist, wodurch sich die Anzahl der Mitglieder ständig ändern kann, entstehen einige Schwachstellen. Diese werden im Folgenden mit den entsprechenden Lösungsmöglichkeiten diskutiert.

In Abbildung 3.1.2 ist einer der Knoten ungenutzt. Er ist am Cachesystem nicht beteiligt. Hinter dem Gedanken sehr begrenzter Ressourcen ist dies pure Verschwendung. Erweitert man das System um eine Restklasse, in dem man 'm' auf drei hoch setzt, müssen alle Daten der bisher zwei Klassen auf den drei Klassen neu verteilt werden. Es entsteht ein großer Verwaltungsoverhead, verbunden mit der Übertragung sämtlicher Cachelines innerhalb des Clusters. Dies verursacht, abhängig von der Menge der gespeicherten Daten, Dienstverzögerungen des Caches. So wie auf Grund der Dynamik jederzeit ein Mitglied einem Cluster beitreten kann, ist auch ein Verlassen möglich. Eine Reorganisation des Clusters ist dann

erneut notwendig. In einer Umgebung mit hochdynamischen Knoten wie Kraftfahrzeuge auf einer Autobahn, die Stau und Straßeninformationen untereinander austauschen, führt dies zu ständigen Reorganisationen. Eine annehmbare Effektivität des Cachesystems ist dann unwahrscheinlich, da es durch die Reorganisationen nicht in der Lage ist, Daten zu verarbeiten. Der Datenverkehr der Reorganisation kann eventuell die Bandbreite für andere Applikationen so weit einschränken, dass sogar die Effektivität des Netzwerkes beeinträchtigt wird. Neben dem hohen Datenverkehr ergibt sich beim Verlust eines Mitgliedes noch ein größeres Problem. Da jeder Knoten einen disjunkten Teil der Daten hält, gehen diese bei Verlust des Knotens mit verloren. Über die Bewegungen der Knoten gibt es keine Kontrolle. Ein vorheriges Reorganisieren auf weniger Restklassen ist dadurch nicht möglich. In dem oben erwähnten Beispiel von Kraftfahrzeugen lässt sich das Ausmaß dieses Effektes deutlich machen. Bezogen auf einen Cluster können die meisten Knoten hier als Transitknoten bezeichnet werden. Sie durchfahren sozusagen den Cluster. Wird in diesem Umfeld wie beschrieben verfahren, verliert der Cluster ständig Daten. Ein Workaround ist hier durch die spezielle Behandlung von Transitknoten möglich. Dabei müssen neue Knoten eine Mindestzeit im Cluster verweilen, bevor eine Reorganisation stattfindet. Hierbei stellt sich allerdings die Frage, ab wann ein Knoten nicht mehr als Transitknoten gilt. Denn selbst wenn er diesen Status erreicht hat, und eine Reorganisation stattfindet, kann er den Cluster unmittelbar danach wieder verlassen. Dieser Ansatz scheint zwar gegen schnelle Transitknoten zu helfen, allerdings besteht das generelle Problem des Datenverlustes weiterhin.

Anstatt die Cachelines unbedingt disjunkt auf allen Knoten zu verteilen, kann man Kopien der Restklassen erzeugen. Das Ziel ist hier **Redundanz** in der Verteilung. Existiert von einer Restklasse immer mehr als ein Replikat, ist die Wahrscheinlichkeit, dass durch Verlust eines Knotens im Cluster Teile des Caches verloren gehen, gleich Null. In der Abbildung 3.1.2a ist der Cluster aus Abbildung 3.1.2 mit redundanter Verteilung dargestellt. Damit von jeder Restklasse mehr als ein Replikat existiert, muss die Anzahl der Restklassen hier auf zwei reduziert werden.

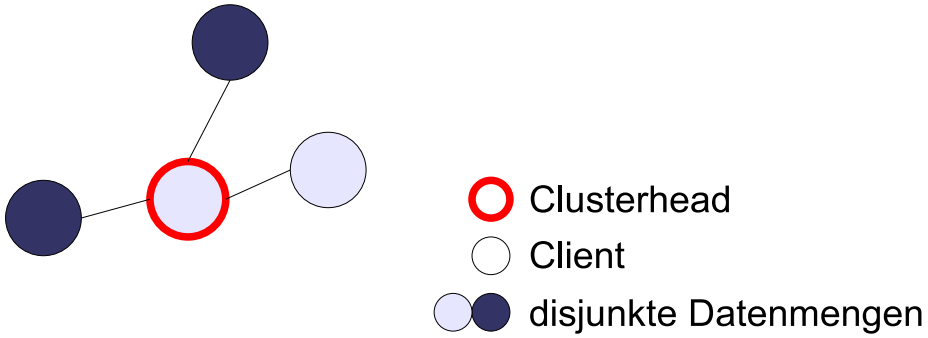


Abbildung 3.1.2a: Hashverteilung mit Redundanz

Da nun von jeder Restklasse zwei Exemplare existieren, hat der Verlust eines Knotens keinen Einfluss auf die Verfügbarkeit der Daten. Wir haben zuvor gefordert, dass immer mehr als ein Replikat jeder Restklasse existiert. Dies ist in dem Beispiel nach Verlust eines Knotens nicht mehr der Fall. Der Verlust eines weiteren Knotens führt dennoch unwillkürlich zu Datenverlust. Um diesen Datenverlust auszuschließen, muss eine Reorganisation stattfinden sobald weniger Replikate als das geforderte

Minimum existieren. Hier ist somit nach Verlust des ersten Knotens eine Reorganisation notwendig. Die resultierende Verteilung besteht dann nur noch aus einer Restklasse. Unter dieser Bedingung entspricht der Cluster der ROWA Verteilung aus Kapitel 3.1.1.

Eine Kombination der Forderung von mindestens zwei Replikaten und dem Wunsch nach maximaler Verteilung ergibt sich für die Anzahl m der Restklassen folgende Beziehung aus der Anzahl n der Clustermitglieder:

$$m = \begin{cases} 1 & n < 2 \\ \left\lfloor \frac{n}{2} \right\rfloor & n \geq 2 \end{cases} \qquad m, n \in \mathbb{N}$$

Bei einer möglichst gleichmäßigen Verteilung der Replikate auf die existierenden Knoten gibt es immer zwei oder drei Replikate einer Restklasse. Das dritte Replikat ist bezüglich der Forderungen ein Überschussreplikat. So lange Überschussreplikate existieren muss die Anzahl der Restklassen nicht reduziert werden. Es kann zwar Situationen mit unterbesetzten Restklassen kommen, doch durch Migration von Knoten mit Überschussreplikaten in diese Restklassen, lässt sich die Redundanz wieder herstellen.

Je mehr Mitglieder ein Cluster hat, um so häufiger ist davon auszugehen, dass sich die Konstellation des Clusters ändert. In einem Cluster mit drei Mitgliedern ist der Verlust von zwei Mitgliedern unwahrscheinlicher als in einem Cluster aus 16 Mitgliedern. Die Gefahr, Daten zu verlieren, steigt demzufolge mit der Größe der Cluster. Die Anzahl der Reorganisationen steigt dabei ebenfalls mit an. Bei grösseren Clustern reduzieren sich somit wieder die Vorteile, die durch die Einführung der Redundanz erworben wurden. Eine Anpassung der Beziehung zwischen Restklassenanzahl und Mitgliederanzahl an steigenden Mitgliederanzahlen gleicht dies wieder aus. Wird die Mindestanforderung an Replikate mit der Anzahl der Mitglieder erhöht, können auch mehrere Mitglieder gleichzeitig den Cluster verlassen, ohne dass Datenverluste auftreten und teure Reorganisationen durchgeführt werden müssen. Im umgekehrten Fall wird die Häufigkeit der Reorganisationen durch Mitgliederzuwachs ebenfalls reduziert. Die neue Beziehung zwischen Restklassenanzahl m und Mitgliederzahl n kann zum Beispiel wie folgt aussehen:

$$m = \lfloor \sqrt{n} \rfloor \qquad m, n \in \mathbb{N}$$

Nun steht die Anzahl der Clustermitglieder in einem quadratischen Verhältnis zur Anzahl der Restklassen. Die Häufigkeit von Reorganisationen bleibt unabhängig von der Größe des Clusters annähernd konstant, was auf die Erhöhung der Redundanz zurück zu führen ist. Ebenso verhält sich auch die Wahrscheinlichkeit von Datenverlusten annähernd konstant. In dem Beispiel aus Abbildung 3.1.2a bleibt die Verteilung mit der neuen n-m-Beziehung bis zu einer Clustergröße von fünf Mitgliedern äquivalent zur alten Beziehung. Bei sechs Mitgliedern wurde vorher die Restklassenanzahl auf drei erhöht. Nun ergibt sich folgender Wert für m:

$$\begin{aligned} m &= \lfloor \sqrt{6} \rfloor \\ &= \lfloor 2{,}44... \rfloor \\ &= 2 \end{aligned}$$

Statt drei Restklassen mit je zwei Replikaten, existieren nun zwei Restklassen mit je drei Replikaten. Bis zu einer Größe von acht Mitgliedern bleibt die Anzahl der Restklassen zwei. Die Redundanz erhöht sich mit Anzahl der Clustermitglieder.

3.1.3 Consistent-Hashing

Im vorherigen Kapitel wurde die Hashverteilung als Lösung für eine disjunkte Verteilung diskutiert. Angepasst an die Dynamik der zu Grunde liegenden Netzwerkstruktur, wurde die Verteilung dynamikfähig gemacht. Es wurde das Problem des erhöhten Datenverkehrs bei Reorganisationen mit Reduktion dieser behandelt. Allerdings muss bei Änderungen der Restklassenanzahl bisher immer die gesamte Datenmenge neu auf die Restklassen verteilt werden. Wird die Reorganisation optimal ausgeführt, muss der gesamte Datenbestand genau einmal übertragen werden. Bei großen Datenmengen wie sie in Netzwerken mit einer großen Anzahl an Clustern entstehen, hat dies eine hohe Netzwerklast zur Folge. Die Übertragung lässt sich dann nicht mit einem einzelnen CellBroadcastpaket realisieren. Der Reorganisationprozess wird dadurch auch stark verlangsamt, was wiederum eine Einschränkung der Verfügbarkeit nach sich zieht. In diesem Kapitel wird eine Lösung für dieses Problem diskutiert.

In [Karger97] wird im Kapitel 4 ein Verfahren namens **Consistent-Hashing** vorgestellt. Es wurde für den Einsatz in verteilten Internet-Proxy-Caches entwickelt. Zu den hervorstechenden Eigenschaften zählen die Möglichkeit mit unterschiedlichen Sichten des Systems arbeiten zu können, die Flexibilität der Verteilung und der geringe Aufwand bei Änderung der Verteilung. Die Verteilungseinheiten in diesem Verfahren sind Buckets. Sie können analog zu den Restklassen der normalen Hashverteilung betrachtet werden. Allerdings müssen Buckets nicht disjunkt sein. Dadurch lässt sich die Handhabung unterschiedlicher Sichten des Systems realisieren. Darauf gehe ich hier allerdings nicht genauer ein, da diese Eigenschaft bei einem clusterweiten Cache nicht von Bedeutung ist. Innerhalb eines Clusters ist eine konsistente Sicht des Clusters garantiert. Vielmehr interessiert der geringe Aufwand bei Größenänderungen. Der Zwang bei der bisherigen Verteilung, alle Daten bei Änderung der Verteilung vollständig neu über den Restklassen zu verteilen, ist in der identischen Bereichsgröße der Restklassen begründet. Die Buckets dagegen müssen nicht in identischen Abständen über dem Wertebereich der Schlüssel verteilt sein. In Abbildung 3.1.3 ist eine derartige Verteilung mit drei Buckets grafisch dargestellt. Die Buckets haben unterschiedliche Abstände. Die grauen Linien unterteilen den Wertebereich in Zuständigkeitsbereiche der Buckets. Die Cachelines werden immer im nächstliegenden Bucket gespeichert.

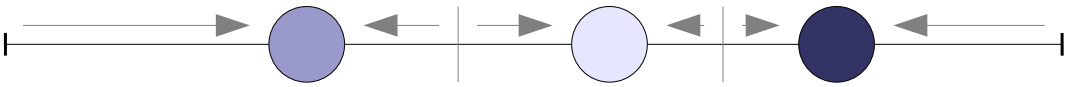


Abbildung 3.1.3: Bucketverteilung

Kommt ein neues Bucket hinzu, wird es auf dem Wertebereich der Schlüssel mittig zwischen zwei Buckets bzw. zwischen einem Bucket und einem Wertebereichsrand oder im Fall des ersten Buckets zwischen den beiden Rändern eingeordnet. Bei

mehreren Möglichkeiten wird anhand des dortigen, momentanen Datenaufkommens entschieden. Die Bucketbereichsgrenzen verschieben sich näher an die vorhanden Buckets heran (siehe Abbildung 3.1.3a). Das heißt, dass die Daten des betroffenen Bereiches von den zwei alten Buckets, beziehungweise dem einen alten Bucket, auf das neue migriert werden müssen. Der Aufwand dafür beläuft sich auf $O(\log n)$ (vgl. [Karger97]).

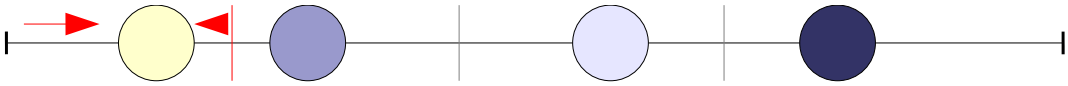


Abbildung 3.1.3a: neues Bucket

In Abbildung 3.1.3a ist zu sehen, dass nur der rot markierte Bereich übertragen werden muss. Die Zuständigkeit für den Rest der Daten bleibt unverändert. Das benötigte Übertragungsvolumen für eine Reorganisation und auch die dafür benötigte Zeit, wird stark reduziert. Eine gleichmäßige Verteilung wie bei der einfachen Verteilung per Hashverfahren kann nicht gewärleistet werden. Allerdings ist dies gar nicht nötig. Es kann angenommen werden, dass die Cachelines sich nicht gleichmäßig auf dem gesamten Schlüsselbereich verteilen. Die Verteilung auf die Buckets ist damit fairer, da die Cachelines abhängig von ihrer Häufung verteilt sind. Grund dafür ist die Anpassung an die momentanen Datenaufkommen bei der Reorganisation.

Durch das hervorragende Verhalten bei Änderungen der Clusterkonstellation stellt sich **Consistent-Hashing** optimal für den Einsatz im Cache des geclusterten Ad-Hoc-Netzes dar. Unabhängig von der Art der Verteilung wird die Redundanz der Buckets wie in Kapitel 3.1.2 realisiert. Folgende Eigenschaften ergeben sich nun zusammengefasst für die Verteilung nach dem Consistent-Hashing:

- faire Verteilung der Daten
- Redundanz
- geringer Aufwand bei Reorganisationen
- geringe Wahrscheinlichkeit für Reorganisationen bei verhältnismäßig kleinen Änderungen des Clusters

Bei der Anwendung des Consistent-Hashing muss ein Fakt berücksichtigt werden. Die Verteilung in Buckets beruht auf einem abgeschlossenen Definitionsbereich. Es muss ein Minimum und ein Maximum existieren. Wird mit einem offenen Definitionsbereich gearbeitet, ist eine Entscheidung, wo genau ein neues Bucket, dass nicht gerade zwischen zwei existierende gehört, platziert wird. Die Schlüssel der Cachelines stellen aber einen offenen Definitionsbereich dar. Hier muss zuerst eine Abbildung auf einen abgeschlossenen Wertebereich durchgeführt werden, um korrekt verteilen zu können. Dies übernimmt wie schon in Kapitel 3.1.2 die Modulooperation. Das Ergebnis der Abbildung ist ein diskreter und begrenzter Wertebereich. Diskrete Werte sind zwar nicht gefordert, aber bei der Implementierung in Kapitel 4 wird es sich als vorteilhaft herausstellen. Die Anzahl der Restklassen ist hier statisch. Eine Restklasse ist die kleinste verteilbare Einheit. Sie stellt die Granularität dar, mit der die Daten auf den Buckets verteilt werden. Somit ist eine Änderung nachträglich nicht mehr möglich.

Im Laufe der Implementierung und Testung, und in Bezug auf mögliche andere Zielsysteme als der PSMW, kann die Anzahl der Restklassen angepasst werden. Für die weitere Entwicklung des Cachesystems wird diese auf 256 festgesetzt. Dieser Wert ergibt sich auch schon aus Sicht der späteren Implementierung, da ein Byte, die Basiseinheit in der elektronischen Datenverarbeitung, einen Bereich von genau 256 diskreten Werten aufspannt. Die Hashfunktion sieht dann wie folgt aus:

$$h(k) \quad = \quad k \bmod 256 \qquad\qquad h(k) \in \{0,\ldots,255\}$$

Die folgenden Betrachtungen ergeben, dass dieser Wert eine gute Wahl ist. Mit handelsüblicher Hardware kann ein Knoten in realer Umgebung eine optimistische Reichweite von ca. 50 Metern erreichen. Hier wird von einer Büroumgebung, mit positiven Eigenschaften für Funknetzwerke ausgegangen, so dass eine optimistische Reichweite erreicht wird. Beansprucht jeder Knoten einen Platz von einem Quadratmeter, können $\lfloor 50^2 \cdot \pi \rfloor - 1 = 7852$ Clients zu einem Cluster gehören. Abgesehen von der Round Trip Länge von $2 \cdot 7852 = 15704$ Slots (siehe Kapitel 2.7), durch die die Kommunikationsfähigkeit lahmgelegt ist, ist die Anzahl von 256 möglichen Buckets noch nicht erreicht. Mit der quadratischen Beziehung zwischen Restklassen und Knotenanzahl, $m = \lfloor \sqrt{n} \rfloor$, werden lediglich 88 Buckets benötigt. Fast jedem Bucket können im Schnitt noch drei Restklassen zugewiesen werden. Eine einigermaßen faire Verteilung anhand von Datenhäufungen ist somit immer noch möglich. Mit 65536 Migliedern im Cluster erreicht das System die maximale Anzahl von 256 Buckets. Erst bei einer Anzahl von 66049 Knoten würde ein 257. Bucket benötigt. Für diesen Fall ist der Cache auf 256 Buckets begrenzt. Es werden dann nur noch weitere Replikate, aber keine neuen Buckets mehr erzeugt. Da bereits das erste Zahlenbeispiel von 7853 Knoten als nicht funktionsfähig eingestuft wurde, sind die letzten Betrachtungen rein theoretisch, um keinen Fall undefiniert zu lassen. Im Rahmen der PSMW sind diese Clustergrößen sogar theoretisch nicht möglich, da sie in Kapitel 2.3 bereits in Hinsicht auf die Kommunikationsfähigkeit auf 16 Mitglieder und somit 15 Clients beschränkt sind. Gegenüber dieser geringen Anzahl an möglichen Clustermitgliedern ist die Anzahl von 256 Restklassen ein feine Granularität, die die Verteilung auf die Buckets bezüglich der Fairness vorteilhaft beeinflusst.

Wie bereits erwähnt, gibt es von jedem Bucket mehrere Replikate, um Redundanz zu erreichen. Da alle Knoten eines Clusters am Cache beteiligt sind, steigt die Anzahl der Replikate pro Bucket linear mit der Anzahl der Buckets an. Dies ergibt sich aus der quadratischen Abhängigkeit der Bucketanzahl. Ein Cluster mit 16 Knoten hat 4 Buckets und je Bucket 4 Knoten, die die Daten eines Buckets halten. Wenn nicht alle Knoten eines Buckets gleichzeitig den Cluster verlassen, tritt kein Datenverlust auf.

In dem folgenden Beispiel wird das Verhalten des Caches in einem dynamischen Umfeld beschreiben. Ausgangspunkt ist ein Cluster mit 7 Knoten (Abbildung 3.1.3.b). Er besteht aus 2 Buckets. Einer mit 3 und einer mit 4 Replikaten. Die Verteilung der Buckets auf dem Definitionsbereich von 0 bis 255 ist in Abbildung 3.1.3c dargestellt.

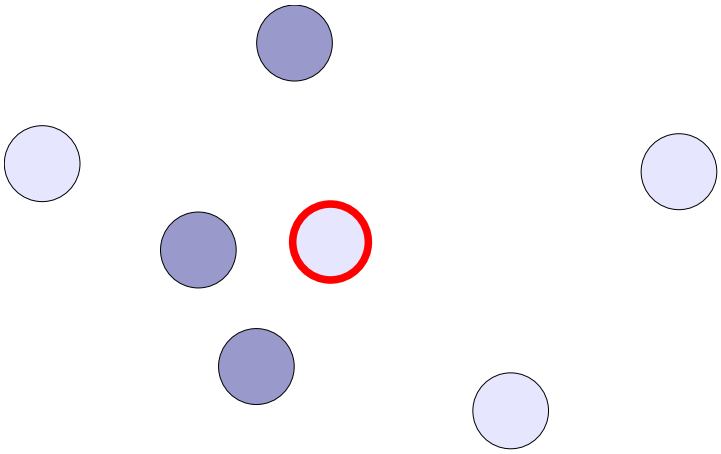


Abbildung 3.1.3b: Ausgangszustand

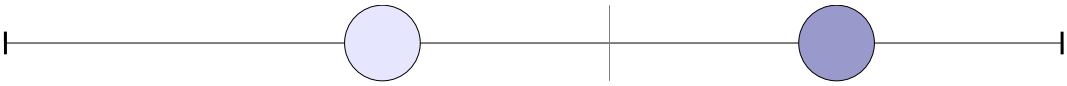


Abbildung 3.1.3c: Bucketverteilung Ausgangszustand

Tritt ein neuer Knoten dem Cluster bei, wird er in das Cachesystem aufgenommen. Nun sind 8 Knoten im Cache. Die Anzahl der benötigten Buckets bleibt bei zwei. Es wird somit kein neuer Bucket erstellt. Der Clusterhead weist den Knoten dem Bucket mit nur drei Replikaten zu. Die Entscheidung wurde anhand der gleichmäßigen Verteilung von Replikaten getroffen. Abbildung 3.1.3d zeigt das Ergebnis mit acht Clustermitgliedern. Die Verteilung der Buckets bleibt unverändert.

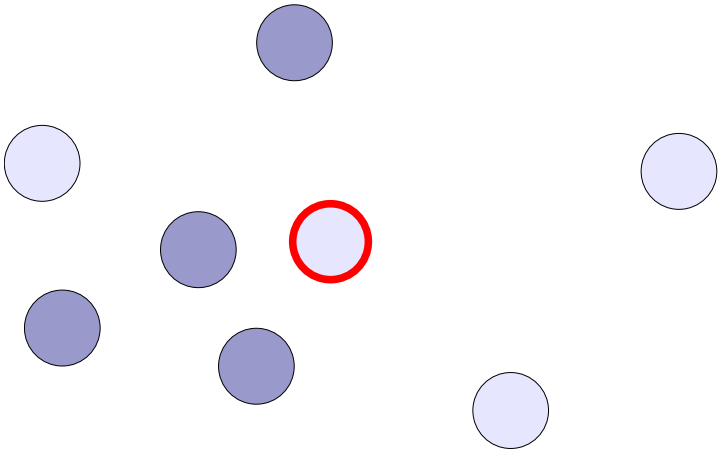


Abbildung 3.1.3d: 8 Mitglieder

Als Nächstes treten zwei neue Knoten dem Cluster bei. Hierbei wird gezeigt, wie sich der Cluster bei Erhöhung der Bucketanzahl verhält. Der Cluster besteht nun aus 10 Knoten. Mit $m=\lfloor\sqrt{10}\rfloor=3$ muss eine Reorganisation durchgeführt werden, da sich die Anzahl der Buckets von zwei auf drei erhöht. In Abbildung 3.1.3e ist dargestellt, das der neue Bucket zwischen den beiden existierende platziert wird. Dadurch verschieben sich die Grenzen dieser der beiden existierenden Buckets. Die entsprechenden Daten werden auf das neue Bucket migriert.

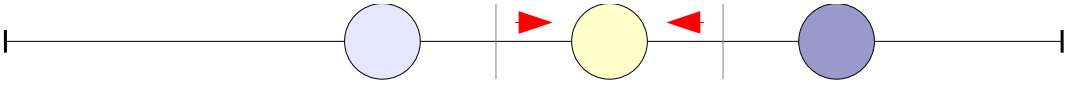


Abbildung 3.1.3e: Verteilung mit 3 Buckets

Die zwei neuen Knoten werden automatisch Replikate des dritten Buckets (gelb). Da die Verteilung der Knoten mit 4-4-2 nun nicht optimal gleichmäßig ist, migriert ein Knoten eines alten Buckets zu dem neuen. Das Ergebnis ist dann 4-3-3 und ist bezüglich der Gleichverteilung optimal. Um den Aufwand einer Reorganisation minimal zu halten, müssen so wenig wie möglich Daten migiert werden. Teile der Daten der beiden existierenden Buckets werden parallel an alle drei Knoten des neuen Buckets übertragen. Angenommen nur einer der neuen Knoten wird statt dem neuen Bucket einem alten zugewiesen, dann müssen zwei Knoten der alten Buckets in den neuen migrieren. Hierfür müssen nun, wie gehabt, Teile der Daten der Buckets übertragen werden. Aber zusätzlich muss für die Migration eines neuen Knotens in eines der beiden alten Buckets der gesamte Datenbestand dieses Buckets übertragen werden. Der Übertragungsaufwand ist deutlich höher. Die Einbindung von nur einem neuen Knoten in eines der existierenden Buckets ist damit suboptimal und sollte somit in dieser Konstellation vermieden werden. Ein mögliches Ergebnis der optimalen Reorganisation ist in Abbildung 3.1.3f zu sehen. Die 10 Knoten des Clusters verteilen sich so gleichmäßig wie möglich auf die Buckets.

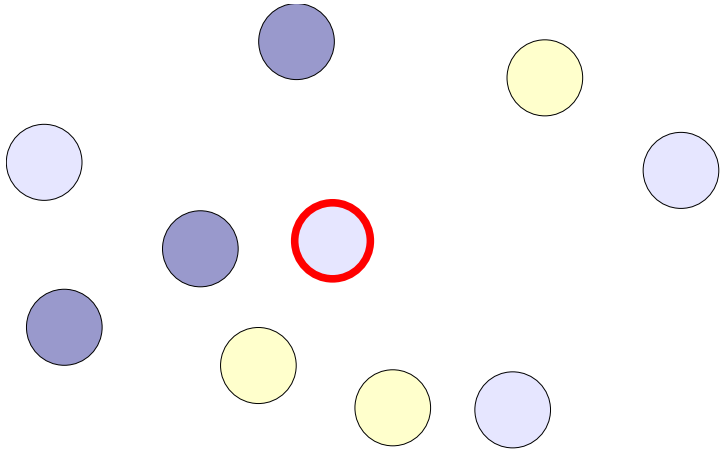


Abbildung 3.1.3f: nach optimaler Reorganisation

Genauso wie ein Cluster wächst, kann er sich auch verkleinern. Analog zum Wachstum verhält sich der Cache dann auch bei Verlust von Knoten. Im nächsten Schritt dieses Beispiels verliert der Cluster einen Knoten. Handelt es sich dabei um einen Knoten des überbesetzten Buckets mit vier Replikaten muss nicht auf dieses Ereignis reagiert werden. Neben der neuen Konstellation des Clusters müssen keine Daten übertragen werden. Die Konstellation lässt sich in einem einzelnen CellBroadcastpaket versenden. Anders ist es, wenn ein Knoten der zwei minmal besetzten Buckets den Cluster verlässt. Nun muss das Gleichgewicht in der Verteilung der Replikate wieder hergestellt werden. Aus einer 4-3-2-Verteilung muss eine 3-3-3-Verteilung entstehen. Dafür migiriert ein Knoten des überbesetzten Buckets mit vier Replikaten zu dem unterbesetzten Bucket mit zwei Replikaten. Hier ist der Aufwand natürlich höher als im vorherigen Fall. Der gesamte Datenbestand des unterbesetzten Buckets muss übertragen werden, damit der migrierende Knoten diesen empfängt. Abbildung 3.1.3g zeigt eine mögliche Konstellation des Clusters, nachdem ein Knoten eines minimal besetzten Buckets entfallen ist.

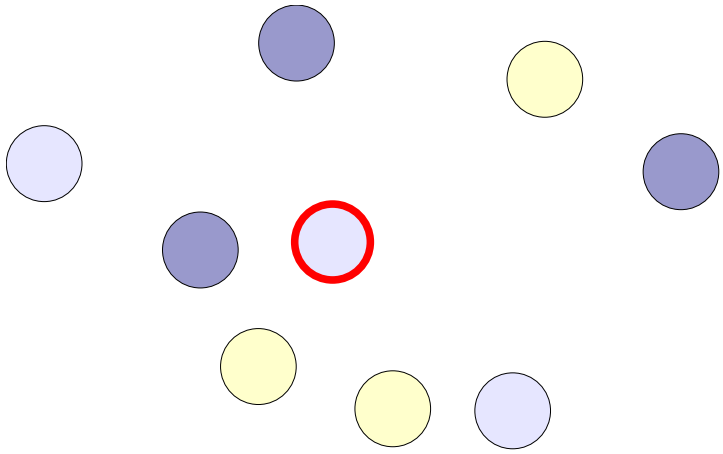


Abbildung 3.1.3g: nach Wiederherstellung des Gleichgewichts

Da der Cluster immer noch aus neun Knoten besteht, ist mit $m=\lfloor\sqrt{9}\rfloor=3$ die Anzahl der Buckets unverändert. Im letzten Schritt des Beispiels verliert der Cluster einen weiteren Knoten. Nun besteht der Cluster nur noch aus acht Knoten, wodurch die Anzahl der Buckets mit $m=\lfloor\sqrt{8}\rfloor=2$ wieder auf zwei sinkt. Ein Bucket muss aufgelöst werden. Für die Entscheidung, welches aufgelöst wird, ist die Unterbesetzung irrelevant. In jedem Fall muss der gesamte Datenbestand des aufzulösenden Buckets übertragen werden, damit die anderen beiden Buckets diese erhalten. Einzig die Auslastung der Buckets ist entscheidend. Das am geringsten ausgelastete Bucket wird demzufolge aufgelöst. Die benötigte Datenübertragung ist dann minimal und die resultierende Verteilung bleibt maximal fair. Wird ein stärker ausgelastetes Bucket aufgelöst, ist die resultierende Verteilung der Cachelines auf die Buckets weniger fair. Vor dem Verlust des Knotens lag eine 3-3-3-Verteilung der Buckets vor. Durch den Knotenverlust resultiert eine 3-3-2-Verteilung. Diese wird durch die Reorganisation zu einer 4-4-Verteilung. In Abbildung 3.1.3h ist ein möglicher Verteilungsbereich mit nur zwei Buckets dargestellt. Abbildung 3.1.3i zeigt eine dazu passende mögliche Verteilung der Knoten auf die Buckets.

Abbildung 3.1.3h: Bucketreduktion

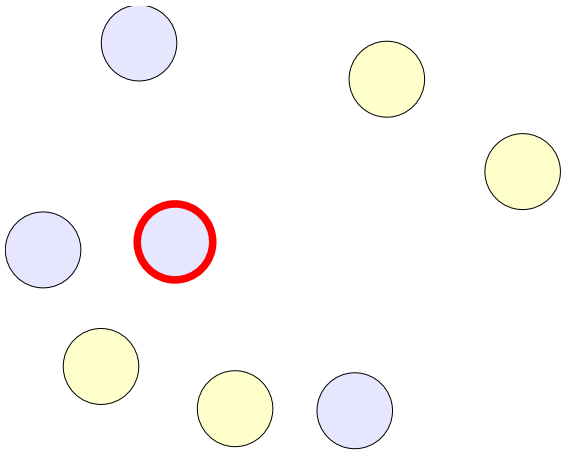


Abbildung 3.1.3i: Replikate nach Bucketreduktion

Zusammengefasst erfüllt die Verteilung mit Consistent-Hashing alle geforderten Eigenschaften optimal. Die Daten sind disjunkt verteilt auf die Mitglieder eines Clusters gespeichert. Diese Verteilung orientiert sich an der momentanen Verteilung der Auslastungen mit Cachelines. Ausfälle von Knoten im begrenzten Rahmen werden durch Redundanz der Daten kompensiert. Reorganisationen stellen bei sich verändernden Clusterkonstellationen immer wieder Zustände her, die gegenüber Ausfällen von Knoten fehlertolerant sind, und dabei eine maximale Verteilung der Datenmengen auf die Clustermitglieder gewährleisten.

3.1.4 Clusterkonfiguration

Die Verteilung, die beschreibt, wie die Restklassen auf die Knoten aufgeteilt sind, heißt Clusterkonfiguration. Die Konfiguration muss als Basis der verteilten Verarbeitung auf allen Knoten eines Clusters konsistent vorliegen. In diesem Kapitel wird erklärt, wie die Konfiguration instand gehalten wird.

Die Konfiguration ist direkt von der Konstellation des Clusters abhängig. Diese Konstellationen ändern sich immer dann, wenn ein oder mehrere Knoten dem Cluster beitreten oder ihn verlassen. Das Cachesystem muss über diese Änderungen informiert werden, um die Verteilung der veränderten Situation anzupassen. Der Cache hat einen Event Handler, der genau die nötigen Aktionen bei Änderungen ausführt. Das Clustering des Netzes wird von der PSMW erledigt. Somit ist diese für die Auslösung des entsprechenden Ereignisses zuständig.

Für den Prozess der Konfiguration gibt es zwei unterschiedliche Möglichkeiten. Beim Push-Verfahren sendet der Clusterhead die neue Konfiguration selbst. Beim Pull-Verfahren dagegen sendet er die Konfiguration nur auf Anforderung der Clients. Im Folgenden werden beide Verfahren untersucht und verglichen. Bei beiden Verfahren ist der Clusterhead für die Verteilung verantwortlich. Wird das Konfigurationsereignis ausgelöst, muss dieser anhand der neuen Konstellation des Clusters eine Verteilung festlegen. Dabei werden alte Verteilungen berücksichtigt. Im nächsten Kapitel wird beschrieben, wie Cluster verglichen werden, um die Verteilungen vorheriger Cluster im neuen Cluster wiederverwenden zu können. Ist die Verteilung fertig, wird sie auf dem Clusterhead angewandt. Um sie auf den Clients anzuwenden, wird sie beim **Push-Verfahren** vom Clusterhead per CellBroadcast versendet. Die Relevanz der Verteilung für die Funktion des Cachsystems fordert bei der Übertragung eine hohe Zuverlässigkeit. Gemessen an der Fehlertoleranz anderer Vorgänge im System können hier keine Fehler toleriert werden. Der CellBroadcast stellt keine Übermittlungsgarantien bereit. Bei dieser einen Übertragung müssen alle Clients die Konfiguration empfangen. In einem Netz ohne Störquellen ist das kein Problem. Aber solch ein Netz existiert nur theoretisch. Gerade unter Verwendung von IEEE802.11 [b|g], das das 2,4GHz ISM Band benutzt, muss von Fehlern in der Übertragung ausgegangen werden. Durch Wiederhohlungen der Übertragung verringert sich die Wahrschheinlichkeit, dass Clients diese nicht empfangen. Nun kann die Größe eines Clusters aber variieren. Bei konstanter Anzahl von Wiederhohlungen steigt die Wahrscheinlichkeit für Übertragungsverluste, wenn die Anzahl der Clustermitglieder steigt. Die Anzahl der Wiederhohlungen muss somit mit der Anzahl der Mitglieder

ansteigen. In Kapitel 3.7 wird dies mit theoretischen Analysen untermauert. Die Clients reagieren beim Push-Verfahren nicht auf das Konfigurationsereignis. Sie reagieren lediglich auf den Empfang einer Konfiguration, in dem sie sie anwenden. Mehr als ein CellBroadcast kann innerhalb eines Slots versendet werden. Da dem Clusterhead in einem Round-Trip so viele Slots zur Verfügung stehen wie Clients im Cluster existieren, benötigt die Konfiguration mit dem Push-Verfahren weniger als einen Round-Trip.

Beim **Pull-Verfahren** ist das Handling des Konfiguartionsereignisses für den Clusterhead mit der Anwendung der Verteilung auf dem eigenen Knoten beendet. Dafür müssen die Clients hier mehr tun. Sie müssen die neue Konfiguration beim Clusterhead anfordern. Dieser beantwortet jede Anfrage mit dem Versenden der Konfiguration. Im Gegensatz zum Push-Verfahren, wo die Wiederhohlung explizit vom Clusterhead behandelt werden muss, ist dies hier implizit gegeben. Die Konfiguration wird durch die einzelne Beantwortung so oft versendet, wie Clients diese angefordert haben. Die zu erwartende Zuverlässigkeit des Konfigurationsprozesses ist damit in beiden Verfahren identisch.

Das Ergebnis beider Verfahren ist identisch. Die Konfiguration wird genauso oft gesendet. Ein Unterschied ergibt sich allerdings bei der Dauer des Konfigurationsprozesses. Beim Push-Verfahren kann innerhalb eines Slots mehr als ein CellBroadcast versendet werden. Die Konfiguration kann somit in weniger als einem Round-Trip durchgeführt werden. Das Pull-Verfahren benötigt dazu länger. Da die Konfiguration immer nur auf Anforderung eines Clients versendet wird, ist es maximal möglich, innerhalb eines Slots des Head einmal die Konfiguration zu versenden. Es wird mindestens ein Round-Trip benötigt, damit alle Clients ihre Anforderung an den Head stellen können. Da der Head einen Slot zwischen jedem Clientslot zur Verfügung hat, können die Antworten innerhalb des selben Round Trips gesendet werden. Pull benötigt somit mindestens ein Round Trip, wogegen Push mit weniger als einem Round Trip auskommen kann. In der Summe der Pakete ist Push ebenfalls besser, da keine Anforderungen benötigt werden.

Die Anstrengungen die Übertragungsverluste zu kompensieren, sind hier doch sehr hoch. Da die Konfiguration eine hohe Relevanz hat, ist die Nutzung einer Übertragung mit Rückmeldungen eventuell effektiver, als unbedingt den günstigen CellBroadcast zu verwenden. Unter den Best-Effort-Methoden, die dem Cache zur Verfügung stehen, hat nur der **Reliable CellBroadcast** die nötigen Eigenschaften. Die PSMW kümmert sich dabei um die erfolgreiche Zustellung des CellBroadcastes. In einem störungsfreien Umfeld empfangen alle Clients das Paket beim ersten Übertragungsversuch. Der Clusterhead erfährt dies durch Bestätigungen der Clients. Damit alle Clients die Bestätigung senden können, wird genau ein Round-Trip benötigt. Der Zeitaufwand ist damit schon grösser als beim Push-Verfahren mit CellBroadcast. Kommen noch Übertragungsfehler oder hohe Auslastungen durch Echtzeitkommunikation ins Spiel, so dass nicht alle Clients das Paket empfangen haben, bzw. eine Rückmeldung senden können, sendet der Head das Paket noch einmal. Um die Bestätigungen der Clients zu empfangen, wird wieder ein gesamter Round-Trip benötigt. Pro Wiederhohlung benötigt der Reliable CellBroadcast somit einen Round-Trip. Gemessen an dem Aufwand des eigentlichen Push-Verfahrens, ist der Aufwand erheblich größer. Mit Reliable CellBroadcast kann zwar eine Effektivität

von 100% erreicht werden, aber in Kapitel 3.7 wird dann gezeigt, dass fast 100% mit CellBroadcast erreicht wird. Der Einsatz des teuren Reliable CellBroadcasts ist somit nicht nutzbringend.

3.2 Clustermatching

Ein Cluster stellt immer ein abgeschlossenes Cachesystem dar. Innerhalb eines Cluster wird wie im vorherigen Kapitel beschrieben, eine Verteilung der Daten gehandhabt. Die Verteilung ist somit abhängig von der Identität eines Clusters. Laut Kapitel 2.3 werden Cluster über die Adresse des Clusterheads identifiziert. In Situationen, bei denen der Clusterhead einen Cluster verlässt, übernimmt ein bisheriger Client dessen Rolle. Aus reiner struktureller Sicht ist die geänderte Identität des Clusters dabei nicht relevant. An der Stelle, wo vorher der Cluster in der Topologie angeordnet wurde, nimmt nun ein anderer Cluster dessen Position ein. Die Konfiguration des Cachesystems im Cluster ist allerdings an die Identität gebunden. Bei einem Wechsel wird in dem neuen Cluster eine völlig neue Verteilung aufgebaut. Wird davon ausgegangen, dass nur der Clusterhead den Cluster verlassen hat, ist durch die Redundanz im Cluster garantiert, dass keine Daten verloren gehen. Im schlimmsten Fall der neuen Verteilung ist es allerdings möglich, dass genau so verteilt wird, dass die entstehende Zuordnungen auf keine der vorhandene Daten auf den Knoten passt. Das Cachesystem hat dann keine Daten. Alle Informationen müssen nun neu von den Applikationen ermittelt werden. Dieser Zustand ist absolut suboptimal und kann so nicht akzeptiert werden. In diesem Kapitel wird eine Lösung dieses Problems über die Ähnlichkeit zwischen alten und neuen Clusterkonfigurationen behandelt.

Die Konfiguration eines Clusters ist gegeben durch die Menge der Mitglieder und der auf ihnen festgelegten Verteilung definiert. In der Betrachtung der Ähnlichkeit von Clustern spielt die Verteilung keine Rolle. Der Vergleich einer alten mit einer neuen Konfiguration geschieht im Cachesystem noch bevor die Verteilung definiert wurde. Der Vergleich der Konfigurationen lässt sich somit auf einen Vergleich der Knotenmengen zurückführen. Die Äquivalenz von zwei Mengen ist folgendermaßen definiert:

$$\begin{aligned} A, B &: \textit{Knotenmengen zweier Cluster} \\ A = B &\Leftrightarrow \forall x (x \in A \wedge x \in B) \end{aligned}$$

Zwei Mengen sind demnach genau dann äquivalent, wenn sie dieselben Elemente besitzen. Der Fall, dass ein Vergleich durchgeführt werden muss, tritt aber nur ein, wenn ein Cluster seinen Clusterhead verliert. Dadurch ist impliziert, dass die Menge der vorherigen Mitglieder nicht identisch mit der Menge der jetzigen Mitglieder ist. Eine Äquivalenzbeziehung kommt somit für den Vergleich nicht in Betracht. Vielmehr ist die Ähnlichkeit zweier Mengen von Interesse. Die Ähnlichkeit von Mengen folgt direkt aus ihrer Distanz zueinander. **Felix Hausdorff** hat die Distanz zweier Mengen wie folgt definiert (vgl. [HD2]):

$$\begin{aligned} h(A, B) &: \textit{gerichtete Hausdorff Distanz} \\ h(A, B) &= max_{a \in A}(min_{b \in B} d(a, b)) \end{aligned}$$

$$\begin{aligned} H(A,B) &: \textit{Hausdorff Distanz} \\ H(A,B) &= max(h(A,B),h(B,A)) \end{aligned}$$

$d(a,b)$ ist dabei die Distanz einzelner Elemente. Dieser ist abhängig vom Typ der Elemente. Der Wertebereich der Hausdorff Distanz entspricht dem Wertebereich der Distanz einzelner Elemente. Beim Vergleich zweier Knoten wird mit folgender Distanz gearbeitet:

$$d(a,b) = \begin{cases} 0 & \Leftrightarrow a=b \\ 1 & sonst \end{cases}$$

Der Wertebereich der Distanz, und somit auch der der Hausdorff Distanz, ist dann $\{0,1\}$. Es gibt nur zwei diskrete Werte, null und eins. Die Hausdorff Distanz hat in diesem speziellen Fall dieselbe Mächtigkeit wie die Äquivalenzdefinition zuvor. Eine Anwendung als Ähnlichkeitsmass ist damit ausgeschlossen.

Ein Maß muss hier einen stetigen Wertebereich haben. Da die Distanz einzelner Elemente aber nur zwei diskrete Werte annehmen kann, bietet sich ein Mittel über die Distanzen der einzelnen Elemente an. Die gerichtete Hausdorff Distanz wird hierbei so verändert, dass nicht das Maximum aller Einzeldistanzen, sondern deren Mittelwert zurückgegeben wird. Der Wertebereich der angepassten, gerichteten Hausdorff Distanz ist stetig von 0 bis 1 ($\{0..1\}$).

$$\begin{aligned} h'(A,B) &: \textit{angepasste , gerichtete Hausdorff Distanz} \\ h'(A,B) &= \frac{1}{|A|}\sum_{a\in A} min_{b\in B} d(a,b) \end{aligned}$$

$$\begin{aligned} H'(A,B) &: \textit{angepasste Hausdorff Distanz} \\ H'(A,B) &= max(h'(A,B),h'(B,A)) \end{aligned}$$

Die resultierende angepasste Hausdorf Distanz hat ebenfalls den stetigen Wertebereich $\{0..1\}$. Für die Betrachtung der Ähnlichkeit kann sie somit genutzt werden. Die Ähnlichkeit wird als Komplement der Distanz definiert.

$$\begin{aligned} S(A,B) &: \textit{Ähnlichkeit zweier Cluster} \\ S(A,B) &= 1-H'(A,B) \\ &= 0 \Leftrightarrow A\cap B=\emptyset \\ &= 1 \Leftrightarrow A=B \end{aligned}$$

Ist der Wert der Ähnlichkeit zweier gegebener Cluster null, ist die Menge ihrer Knoten elementfremd. Es gibt keine Übereinstimmungen bei ihren Mitgliedern. Ist dagegen der Wert der Ähnlichkeit eins, handelt es sich um äquivalente Knotenmengen. Die Cluster sind bezüglich ihrer Mitglieder identisch.

Da nun ein Ähnlichkeitsmaß vorliegt, muss definiert werden, bis zu welchem Wert zwei Cluster als identisch angenommen werden. An dieser Stelle ist die Frage relevant, bis zu welcher Distanz die Verteilung eines alten Clusters von einem neuen erfolgreich übernommen werden kann. Ein Erfolg ist dadurch gekennzeichnet, dass mindestens ein Knoten, der sowohl in dem alten als auch in dem neuen Cluster vorhanden ist, die Zuständigkeit für mindestens eine Restklasse beibehält. Somit können bestehende Daten übernommen werden. Dies trifft genau dann zu, wenn mindestens ein Knoten in beiden Clustern existiert. Die Ähnlichkeit ist dann

$S(A,B)>0$.

Knoten können zu mehr als einem Cluster gehören. Als Folge dessen ist es möglich, dass ein neuer Cluster ähnlich zu mehr als einem alten Cluster ist. Die Wahl fällt dann auf den Cluster mit der höchsten Ähnlichkeit. Eine maximale Wiederverwendbarkeit der Daten kann dort erwartet werden. Da die Verteilung immer vom Clusterhead definiert wird, und dieser nur exklusiv für einen Cluster zuständig ist, kann immer nur ein neuer Cluster existieren. Ein Vergleich der Ähnlichkeiten mehrerer Kombinationsmöglichkeiten ist somit ausgeschlossen.

3.3 Ersetzungsstrategie und Alterung

Die Caches besitzen einen begrenzten Speicher. Daher müssen die Cachelines auch ersetzt werden, wenn neuere hinzugefügt werden sollen. Eine gut durchdachte Ersetzungsstrategie ist gerade in diesem Umfeld relevant, da die Zielsysteme, im Allgemeinen eingebettete Systeme, nur knapp mit Ressourcen ausgestattet sind. Der verfügbare Arbeitsspeicher reicht von ca. 512 kByte bei mobilen Sensoren über Handys und PDAs mit durchschnittlich 4 bis 64 MByte bis hin zu 1 GByte bei Notebooks und immobilen Computersystemen. Letztere stellen hierbei die Ausnahme dar. Sie gehören auch nicht zu eingebetteten Systemen. Ihre Beteiligung ist aber gerade im Anwendungsfeld der mobilen Sensoren durchaus wahrscheinlich, da sie dort als Subscriber zum Auslesen der Sensordaten verwendet werden können.

Ist nicht mehr genügend Speicher zur Aufnahme neuer Cachelines vorhanden, müssen welche gelöscht werden. Die Verwendung einer FIFO Queue ist hier nicht angemessen. Unabhängig vom Nutzen der Cachelines werden sie hier in der Reihenfolge, wie sie aufgenommen werden, auch wieder gelöscht. Häufig abgefragte Cachelines können somit gelöscht werden, obwohl welche existieren, die eventuell noch nie abgefragt wurden. Dieses Verhalten ist suboptimal und dadurch nicht akzeptabel.

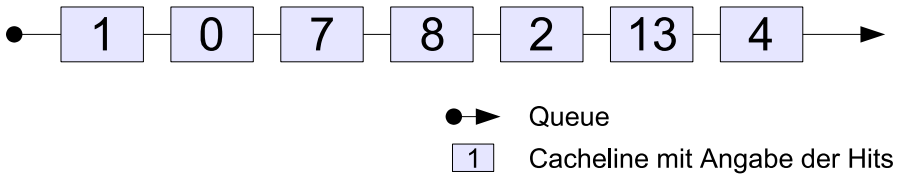


Abbildung 3.3: FIFO Queue

Abbildung 3.3 zeigt eine grafische Darstellung eines sehr kleinen Beispiels. Die FIFO Queue ist von links nach rechts abgebildet. Wenn links eine neue Cacheline eingefügt wird, muss rechts, am Ende der Queue, zuvor eine gelöscht werden. Die Reihenfolge der Ersetzung der Cachelines ist somit von rechts nach links abzulesen. In den Boxen, die die Cachelines darstellen, sind deren Hits eingetragen. Deutlich wird hier die Unabhängigkeit der Ersetzungsstrategie vom Nutzen der Cachelines. Die Anzahl der Hits erhöht sich im Allgemeinen mit der Dauer, die eine Cacheline existiert. Im Schnitt befinden sich dann die meist genutzten Cachelines am Ende der Queue, und werden somit als erstes ersetzt. Eine Ordnung anhand der Nutzung muss eingeführt werden. Ein Ansatz dafür bildet die Häufigkeit der Zugriffe. Um die Effizienz des Caches zu maximieren, müssen die Anfragen erfüllt werden, die am

häufigsten benötigt werden. Die **Effizienz eines Caches** ist durch die Wahrscheinlichkeit definiert, mit der Anfragen nach Daten, die bereits eingetragen wurden, einen Hit hervorrufen. Das heißt, mit welcher Wahrscheinlichkeit Daten über den Cache wieder abgerufen werden können. Zu beachten ist dabei die unterschiedliche Relevanz einzelner Daten. Hintergrund dieser Effizienzdefinition ist die Reduzierung von Anfragen an den Informationsursprung. Befinden sich wenig genutzte Informationen nicht im Cache, wird die Effizienz nur minimal reduziert, da sie in der Gesamtanzahl von Anfragen nur einen kleinen Teil ausmachen. Eine bessere Ordnung für die Ersetzung ergibt sich somit wie in Abbildung 3.3a über die Zugriffshäufigkeiten (Hits). Die Beispielcachelines aus Abbildung 3.3 erscheinen dann in geänderter Reihenfolge. Cachelines mit vielen Zugriffen befinden sich nun vorn in der Queue und diejenigen, mit wenigen Zugriffen, analog dazu hinten in der Queue. Von der Ersetzung sind somit immer die selten genutzten Cachelines betroffen. Die Effizienz des Systems wird erhöht.

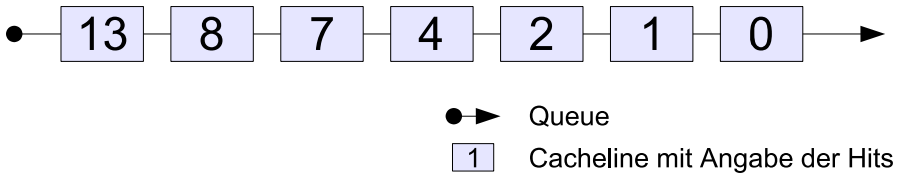


Abbildung 3.3a: Ordnung über die Hits

Ein Problem dieses Verfahrens ist die Stabilisation häufig genutzter Cachelines am Anfang der Ordnung. Ist die Queue einmal gefüllt, bleiben Cachelines, die bis dahin viele Hits hatten, im vorderen Bereich. Neue Cachelines sind zu Beginn hinten in der Queue. Sie können aber nicht mehr an den Anfang wandern, da sie sehr wahrscheinlich ersetzt werden, bevor sie genügend Hits verzeichnen. Durch die Dynamik des Netzes ist es wahrscheinlich, dass Informationen, die anfangs häufig abgefragt wurden, ihre Relevanz verlieren. Sie könnten also ersetzt werden. Da sie sich aber am Anfang der Ordnung stabilisiert haben, geschieht dies nicht. Speicherplatz wird verschwendet und die Effizienz des Systems ist stark beeinträchtigt.

Ein besseres Verfahren nimmt somit Rücksicht auf den Zeitpunkt des letzten Zugriffs. Im **Least Recently Used** (LRU) Verfahren ist dies der Fall. Die Ordnung der Ersetzungsqueue entsteht hierbei über den Zeitpunkt der letzten Nutzung. Am Anfang der Queue stehen immer die zuletzt genutzten Cachelines, und am Ende die, deren Nutzung am längsten her ist. Dieses Verfahren stellt sich als optimal heraus, da die temporäre Abhängigkeit der Informationen hier implizit beachtet wird. Eine Stabilisierung von veralteten Cachelines am Anfang der Queue wird verhindert. Über die Zeit betrachtet wandern Cachelines ständig in Richtung Ende der Queue. Durch einen Hit werden sie wieder an den Anfang versetzt. Neue Cachelines befinden sich zuerst am Anfang der Queue, wandern aber, wenn sie nicht wieder abgefragt werden, zum Ende. Die Relevanz der Informationen, gemessen am Zeitpunkt des letzten Zugriffs, steht nun im Mittelpunkt der Ersetzungsstrategie. Abbildung 3.3b zeigt eine mögliche Ordnung zum Beipiel aus Abbildung 3.3 und 3.3a. Es gibt hier keinen direkten Bezug zu den sieben Daten. Zur Veranschaulichung ist lediglich die selbe Anzahl an Cachelines gewählt. Anstatt wie bisher die absolute Häufigkeit der Hits zu jeder Cacheline anzugeben, ist der letzte Zugriffszeitpunkt bezüglich der

momentanen Zeit dargestellt. Zur Verdeutlichung, dass es sich hier nicht um das Alter einer Cacheline handelt, ist dieses als Balken auf jeder Cacheline angegeben. Für das Maß der Zeit ist hier eine abstrakte Einheit gewählt.

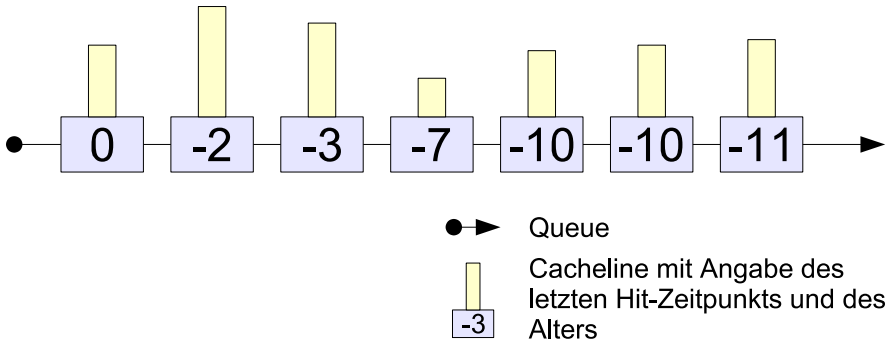


Abbildung 3.3b: LRU

In Umgebungen mit zyklischen Anfragen an einen Cache mit LRU Ersetzungstrategie können Situationen entstehen, bei denen die Effizienz auf 0% sinkt. Dabei werden unterschiedliche Daten in einem Zyklus angefragt, der mindestens eine Anfrage mehr enthält, als Cachelines im Cache gespeichert werden können. Es ergibt sich dann kein einziger Hit, da die Cachelines spätestens eine Anfrage zuvor durch das LRU-Verfahren ersetzt wurden. Grund für diesen Effekt sind zwei Fakten, die für den speziellen Anwendungsfall in dieser Diplomarbeit nicht zutreffen. Cachesysteme, in denen dieser Effekt auftritt, müssen die Anfrage und bei einem **Miss** das Ermitteln der Information vom Ursprung und das Einfügen in den Cache synchron ausführen. Dies ist hier nicht gegeben, da fast alle Funktionen, die hier benötigt werden, auf die Kommunikation im Netzwerk basieren. Die ereignisbasierte Verarbeitung im Netzwerk ist hochgradig asynchron. Durch die Beschränkung auf Best-Effort-Traffic wird diese Asynchronität noch verstärkt, da unterschiedliche Verzögerungen bei der Datenübertragung durch bevorzugte Echtzeitkommunikationen möglich sind. Der zweite Fakt ist durch die benötigten zyklischen Anfragen gegeben. Die Arbeitsweise des zu entwickelnden Cachesystems ist sporadisch. Es handelt sich um ein verteiltes System. Die Ersetzung von Cachelines wird somit nicht nur durch Eintragungen der anfragenden Applikation, sondern auch durch Propagationen von Cachelines anderer Knoten beeinflusst. Zyklische Anfragen sind in diesem Projekt somit als unwahrscheinlich einzustufen.

Nur die Anwendung von LRU hat aber trotzdem noch einen Haken. Da häufig genutzte Cachelines nur mit sehr geringer Wahrscheinlichkeit ersetzt werden, sind sie sehr inaktuell. Gegenüber der offensichtlich hohen Relevanz dieser Informationen, ist dieser Zustand nicht tragbar. Das Netz arbeitet dann mit veralteten Daten. Die Aktualität weniger relevanter Informationen ist dabei im Allgemeinen höher. Kann eine gesuchte Cachelines nicht im Cache gefunden werden, wird die Informationsquelle befragt. Das resultierende Ergebnis ist nun absolut aktuell. Es wird dann erneut im Cache aufgenommen. In Abbildung 3.3b ist dieser Effekt andeutungsweise bei den ersten drei Cachelines dargestellt. Sie sind älter als die weniger genutzten Cachelines. Ohne eine Kontrolle des Alters der Cachelines kommt es bei den häufig genutzten Cachelines nie zu einer Aktualisierung durch Anfrage beim Informationsursprung. Das System befindet sich dann in einer Sackgasse. Gelöst wird das Problem durch die Definition eines maximalen Alters. Bei jedem

Zugriff, jeder Verarbeitung einer Cacheline, wird dann das Alter überprüft. Hat diese das maximale Alter überschritten, wird sie für die Weiterverarbeitung ignoriert und gleich gelöscht. Das System wird somit gezwungen, in festgelegten Abständen die Daten zu aktualisieren. Cachelines, die älter als das maximale Alter sind, können somit nicht existieren. Ein globales Maximalalter, dass für alle Cachelines zutrifft, ist hier ungünstig. Das Cachesystem arbeitet unabhängig von dem Inhalt der Cachelines. Verschiedene Anwendungen können auch unterschiedliche Anforderungen an das maximale Alter stellen. Das Maximalalter wird deshalb je Cacheline behandelt. Die Definition des maximalen Alters wird dabei von den Anwendungen selbst übernommen.

Eine unmittelbare Folge der Ersetzung von Cachelines ist, dass bei jedem Eintragen einer neuen Cacheline in einen vollen Cache erst eine Cacheline gelöscht wird. Dies bedeutet zusätzlichen Aufwand bei jeder Eintragung. Der lässt sich aber vermeiden, indem ein Teil des verfügbaren Speichers für Einfügungen freigehalten wird. Wird dieser freigehaltene Bereich bei jeder Einfügung überprüft, ist nichts gewonnen. Der selbe Aufwand wird benötigt, da immer noch zuerst gelöscht werden muss. Es wird lediglich nicht der gesamte verfügbare Speicherplatz ausgenutzt. Eine periodische Überprüfung des frei zu haltenden Speichers bewirkt die erhofte Verbesserung. Einfügungen können durchgeführt werden, ohne dass erst gelöscht werden muss. Füllt sich der Speicher innerhalb des Zeitfensters zwischen zwei Überprüfungen, wird wie zuvor verfahren. Eine optimierte Parametrisierung des minimal freien Speichers und der Periodendauer der Überprüfungen minimiert den Aufwand bei Einfügungen in den Cache. Diese Optimierung ist von Umgebungsvariablen, wie dem verfügbaren Speicher des Cachesystems und der Häufigkeit von neuen Cachelines, abhängig. Eine Anpassung ist somit erst für Erwartungswerte gezielter Umgebungen möglich.

3.4 Synchronisation

Synchronisation ist immer dann nötig, wenn sich die Zuständigkeit eines Knotens ändert. Dies geschieht bei Reorganisationen des Clusters und bei Änderungen der Bucketzugehörigkeit eines Knotens. Allgemeiner lässt sich sagen, dass bei jeder Zuweisung zu einer Restklasse, der kleinsten Verteilungseinheit, für die ein Knoten zuvor nicht zuständig war, eine Synchronisation dieser Restklasse nötig ist. Das Wissen, ob Knoten existieren, die eine Synchronisation für die Restklasse durchführen können, ist nicht vorhanden. Selbst wenn über die vorherige Verteilungskonfiguration Knoten ausgemacht werden können, die für diese Restklasse schon vorher zuständig waren, kann keine Aussage darüber getroffen werden, ob diese synchronisiert waren. Dieses Kapitel befasst sich mit den genauen Bedingungen, Anforderungen und Lösungen.

In den folgenden Betrachtungen wird ein Knoten, der Daten für die Synchronisierung einer Restklasse benötigt, als **Empfänger** bezeichnet. Analog dazu ist ein Knoten, der Daten für die Synchronisation versendet, ein **Sender**.

Die gesamte Kommunikation innerhalb eines Clusters ist durch die Beschränkung auf CellBroadcast verbindungslos. Dies stellt bei der Synchronisierung das primäre

Problem dar. Bei einem Synchronisationsprozess wird im Allgemeinen mehr als eine Cacheline übertragen. Selbst bei Bündelung mehrerer Cachelines in ein Paket ist davon auszugehen, dass in den meisten Fällen mehr als ein Paket benötigt wird, um die Daten einer Restklasse zu übertragen. Ein Empfänger ist bei dieser Kommunikation nicht in der Lage, Anfang und Ende einer Synchronisation zu erkennen. Jedes Paket stellt für sich eine einzelne Übertragung dar. Die Erkennung, ob ein Sender verfügbar ist, kann noch durch ein Time-out ermittelt werden. Aber die Erkennung des Synchronisationsendes stellt ein weitaus größeres Problem dar. Die Verwendung eines Time-outs bringt hier keinen Erfolg. Wird innerhalb dieses Time-outs die Synchronisation erneut begonnen, weil ein anderer Knoten ebenfalls synchronisiert werden muss, ist die Synchronisation für den Empfänger noch nicht beendet, obwohl bereits alle Daten übertragen wurden. Für die Synchronisation muss eine verbindungsorientierte Datenübertragung auf Applikationsebene, also vom Cachesystem selbst, entwickelt werden. Abbildung 3.4 zeigt den zeitlichen Ablauf der Synchronisation. Anhand der dort aufgezeigten vier Knoten wird die Funktionsweise in den folgenden Absätzen erläutert und begründet.

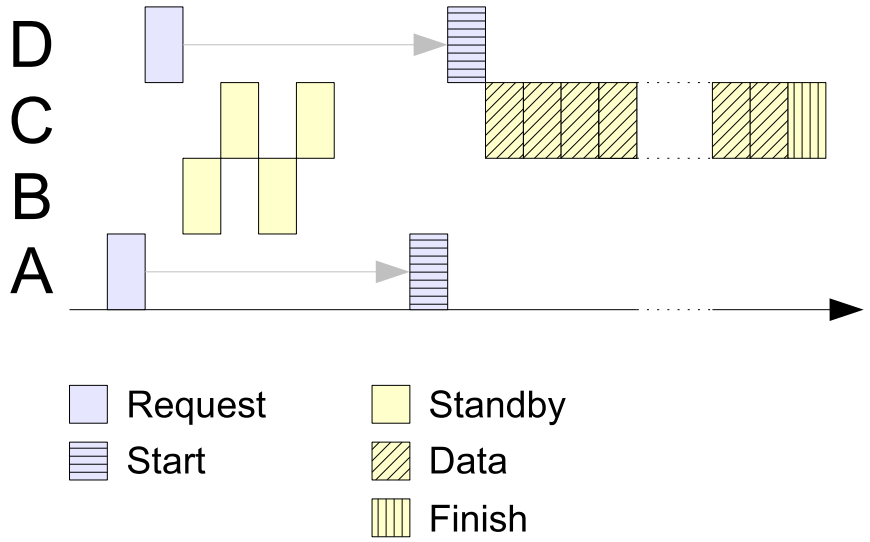


Abbildung 3.4: Synchronisationsprotokoll

Die Synchronisation wird immer vom Empfänger initialisiert. Grund dafür ist die zuvor erwähnte Bedingung, wann eine Synchronisation ausgeführt werden muss. Ein Knoten weiß direkt durch die Zuweisung einer neuen Restklasse, dass er versuchen muss, diese zu synchronisieren. Ein Knoten ist dagegen aber ohne Kommunikation über das Netz nicht in der Lage zu entscheiden, ob ein anderer Knoten im Cluster Daten für die Synchronisierung einer bestimmten Restklasse benötigt. Jeder Empfänger versendet somit als erstes eine Synchronisationsanfrage. Unter der Annahme, dass keine Übertragungsfehler auftreten, wird diese, da CellBroadcast verwendet wird, von allen Knoten des Clusters empfangen. Potentielle Sender, also Knoten, die für die gewünschte Restklasse zuständig sind, und selbst nicht synchronisieren müssen, senden darauf ihre Bereitschaft zur Synchronisation. In Abbildung 3.4 sind A und D Empfänger. Zu beachten ist hier, dass innerhalb eines Clusters immer sichergestellt ist, dass nur ein Knoten sendet. Deshalb wird in der Abbildung auch immer nur genau ein Paket zu einem Zeitpunkt übertragen. Innerhalb des Round Trips war A zuerst in der Lage zu senden, dadurch verschickt er auch

zuerst die Anfrage. Zu beachten ist, dass die Darstellung in Abbildung 3.4 sich nicht direkt auf die Zeitslots aus Kapitel 2.7 bezieht. Jegliche Zwischenschritte des Round-Trips sind der Einfachheit halber nicht dargestellt. Für die Antwort eines Senders ist ein Time-out definiert. Es muss ausreichend groß definiert werden, da die Übertragung der Best-Effort-Pakete durch Echtzeitübertragungen sehr verzögert werden kann. Unter Bedingungen, die für das Cachesystem optimal sind, das heißt, wenn nur der Cache Daten überträgt, wird mindestens ein Round-Trip benötigt, damit jedes Mitglied eines Clusters die Möglichkeit hat, ein Standby zu versenden. Dies ist aber nur ein Idealfall. Verzögerungen über mehrere Round-Trips sind wahrscheinlicher. Der Anfrage Time-out muss somit ein Vielfaches eines Round-Trips sein. Denkbar ist ein Wert von ungefähr 10 Round-Trips. Ein akzeptabler Wert muss in Tests mit realistischen Bedingungen ermittelt werden. In Abbildung 3.4 ist der Time-out beispielhaft als grauer Pfeil dargestellt. Der Anfrage-Time-out hat aber noch eine zweite Aufgabe. Er sorgt dafür, dass die Synchronisation nicht so schnell wie möglich gestartet wird. Eine verzögerter Start der Synchronisation ermöglicht, dass Empfänger, die leicht später dem Cluster beitreten, sich der Synchronisation anschließen können. Dadurch wird eine gehäufte Neuübertragung der Synchronisationsdaten verhindert. Die Synchronisation mehrerer Empfänger wird somit auf einen Vorgang konsolidiert. Es handelt sich somit nicht um einen eigentlichen Time-out. Vielmehr warten die Empfänger nach dem Versenden der Anfrage eine festgelegte Zeit. Danach werden Bereitschaftspakete potentieller Sender, die innerhalb der Wartezeit empfangen wurden, ausgewertet. Wird innerhalb der Wartezeit keine Bereitschaft empfangen, geht der Empfänger davon aus, dass kein Sender verfügbar ist. Die Synchronisation ist dann für den Empfänger beendet. Er setzt den Status der Restklasse auf synchronisiert.

B und C sind potentielle Sender. Nach dem Empfang einer Anfrage senden sie ihre Bereitschaft zur Synchronisation. Bei Empfang mehrerer Anfragen für dieselbe Restklasse senden sie auch mehrere Bereitschaftspakete. So wird implizit eine Redundanz in der Übertragung erzeugt, wie sie bei den Anfragen eventuell explizit nötig ist. Im Bereitschaftspaket wird eine zufällig gewählte Nummer mitgeschickt. Anhand dieser Nummer wird später von den Empfängern der Sender ausgewählt, der die Synchronisation durchführen soll. Eine Wahl ist hier nötig, da der Umfang der zu sendenen Daten weit höher ist, als bei den Anfrage- und Bereitschaftspaketen. Eine Redundanz der Datenpakete sollte deshalb vermieden werden, um nicht zu viel Bandbreite zu benötigen. Eine Synchronisation mit redundanten Sendern ist nicht realisierbar. Als Sender kommt immer nur genau ein Knoten in Frage, da die Daten von zwei Sendern leicht unterschiedlich sein können, und sie selbst bei einem indentischen Stand die Daten in unterschiedlicher Reihenfolge versenden können. Die Synchronisation ist somit an die Identität des Senders gebunden. Da davon auszugehen ist, dass immer mehr als eine Restklasse synchronisiert werden muss, die zum selben Bucket gehören, wird durch die zufällige Nummer im Bereitschaftspaket vermieden, dass ein und derselbe Knoten alle Restklassen synchronisiert. Da das System Inkosistenzen der Daten toleriert, können verschiedene Sender auch leicht verschiedene Daten haben. Durch die zufällige Auswahl wird zusätzlich eine bessere Verbreitung der Daten gewährleistet, da selbst potentielle Sender während einer Synchronisation die Daten eines anderen Senders verarbeiten. In Abbildung 3.4 ist zu sehen, dass B und C zweimal ihre Bereitschaft

senden. Dies ist auf den Empfang der Anfragen von A und D zurückzuführen. Es werden somit insgesamt vier Bereitschaftspakete versendet.

Nach Ablauf der Wartezeit werten die Empfänger die empfangenen Bereitschaften aus. Die Wahl fällt auf den Sender mit der geringsten Zufallszahl. Die Empfänger senden nun ein Startpaket mit der Identität des gewählten Senders. Jeder Empfänger weiß durch das Startpaket, dass im Folgenden Daten für die Synchronisation übertragen werden. Der Anfang einer Synchronisation ist somit definiert. In Abbildung 3.4 hat A zuerst das Senden einer Anfrage initialisiert. Der Anfrage Time-out läuft für A somit zuerst ab. Im Bereitschaftspaket von C hat A eine geringere Zufallszahl empfangen als von B. A sendet somit ein Startpaket mit der Identität von C. Kurz danach läuft auch für D der Time-out ab, und D versendet ebenfalls ein Startpaket mit der Identität von C.

Durch den Empfang des Startpaketes beginnt der Sender, alle Cachelines einer Restklasse zu senden. Dabei wird jeder Cacheline eine Sequenznummer zugewiesen. Über diese Sequenznummer sind die Empfänger später in der Lage, die Anzahl verlorengegangener Cachelines zu ermitteln. Die Cachelines werden soweit möglich gebündelt in einzelnen CellBroadcastpaketen versendet. Hat der Sender alle Cachelines versendet, schickt er ein Schlusspaket. Mit diesem Schlusspaket wird das Ende der Synchronisation gekennzeichnet. Das Ende einer Synchronisation ist somit ebenfalls genau definiert. Die Empfänger zählen während der Synchronisation die Anzahl verlorener Cachelines anhand der mitgeschickten Sequenznummern. Mit dem Sclusspaket erhalten die Empfänger nochmal die Gesamtzahl versendeter Cachelines. Jeder Empfänger kennt nun seine Fahlerrate dieses Synchronisationsprozesses. Wird eine bestimmte Schranke dieses Wertes überschritten, wird der Synchronisationsprozess vom betroffenen Empfänger erneut gestartet. Bei den Wiederholungen der Synchronisation wird die Schranke der akzeptierten Fehlerrate nach oben korrigiert, damit keine unendlichen Wiederholungen der Synchronisation möglich sind. Mögliche Schrankenwerte für die Fehlertoleranz der Synchronisation sind z.B. 10%, 30% und 100%. Durch die 100% Fehlertoleranz in der zweiten Wiederholung wird die Anzahl der Wiederholungen auf zwei begrenzt, da 100% immer erreicht werden.

Während der Synchronisation kann es passieren, dass der Sender nicht mehr verfügbar ist. Um diesen Fall behandeln zu können, wird während des gesamten Synchronisationsprozesses ein Daten-Time-out benutzt. Nach jedem Paketempfang wird dieser neu initialisiert. Läuft der Time-out ab, gilt die Synchronisation als nicht erfolgreich abgeschlossen. Der Empfänger beginnt den Synchronisationsprozess komplett von vorn.

Da die gesamte Kommunikation mit Best-Effort-Traffic arbeitet, ist sie gegenüber Übertragungen mit Dienstgütegarantien geringer priorisiert. Mit Verzögerungen der Datenpakete muss hier gerechnet werden. Da dies von den realen Belastungen der PSMW abhängt, können die verwendeten Time-outs auch nur später bei der Integration und Testung erfolgreich parametrisiert werden.

Unter der Annahme verlustfreier Übertragungen ist mit dem soeben beschriebene Synchronisationsprotokoll der vollständige Abgleich aller Empfänger mit den Daten eines Senders garantiert. Auf diese einfache Weise lässt sich die Synchronisation

koordinieren, ohne dass höhere Funktionen der Netzwerkschicht benötigt werden. Probleme entstehen hier, wenn Übertragungsfehler berücksichtigt werden. In der Übertragung der Daten ist bereits eine akzeptierte Fehlerrate definiert, mit der eine Synchronisation als erfolgreich angenommen wird. Bei den Kontrollpaketen kann eine solche Fehlerrate nicht definiert werden, da es keine Rückmeldungen über deren Empfang gibt, und es dort einzelne Pakete, und nicht eine Menge von Paketen betrifft. Mit Hilfe von Redundanz kann die Zuverlässigkeit der Übertragungen erhöht werden. Dies ist bei den Kontrollpaketen von großer Bedeutung, da hier der Verlust eines einzelnen Paketes den Fehlschlag der gesamten Synchronisation zur Folge haben kann. Bei Existenz nur eines Empfängers ist die Synchronisation von genau einem Anfrage- und einem Startpaket abhängig. Bei der Existenz von nur einem potentiellen Sender ist die Synchronisation von genau einem Bereitschaftspaket abhängig. Noch einschneidender ist die Abhängigkeit der Synchronisation von genau einem Schlusspaket. Ohne dieses ist der erfolgreiche Abschluss einer Synchronisation nicht möglich. Dabei ist die Anzahl der Empfänger und der potentiellen Sender sogar unwichtig. Da genau ein Sender Synchronisationsdaten versendet, wird auch nur genau ein Schlusspaket versendet. In Kapitel 3.7 wird die verlustbehaftete Effektivität stochastisch analysiert. Deshalb wird dies hier nicht weiter betrachtet. Als Lösung dieses Problems wird in Abhängigkeit von den dort erzielten Zuverlässigkeiten der einzelnen Übertragungen, eine Redundanz dieser Pakete durch Übertragungswiederhohlungen erzeugt. Ziel ist dann eine Erhöhung der Zuverlässigkeit bis zu einem bestimmten Maß. Eine Zuverlässigkeit von ca. 95% sollte dabei erreicht werden.

3.5 Propagation

Die soeben beschriebene Synchronisation entspricht der Verbreitung und Replikation der Daten im „offline“-Zustand. Die Daten werden dort unabhängig von der aktuellen Verarbeitung der Cachelines verbreitet. Alle gespeicherten Cachelines werden dazu übertragen. Die Propagation dagegen befasst sich mit der „online“ Verbreitung. Gegenstand sind hier die Änderungen der Daten. Neue und geänderte Cachelines werden im Netz verbreitet. Neben der clusterinternen Verbreitung wird auch über die Clustergrenzen hinweg propagiert. Die Propagation ist somit für die Diffusion der Daten in angrenzende Cluster zuständig. Zuerst wird die Funktionsweise der Propagation generell erörtert. Anschließend folgt der spezielle Fall der clusterübergreifenden Propagation.

Propagationen werden immer dann nötig, wenn Änderungen an den Datenbeständen eines Knotens auftreten. Diese Änderungen können unterschiedlichen Ursprungs sein und lassen sich danach klassifizieren. Zum Einen können Cachelines hinzugefügt werden. Diese müssen dann propagiert werden, damit alle zuständigen Knoten diese auch verarbeiten. Des weiteren können Cachelines inhaltlich aktualisiert werden. Geht man bei Nichtexistenz einer Cacheline von einem Äquivalent zu einer leeren Cacheline aus, handelt es sich in den beiden Fällen um dieselbe Art der Änderung. Die erste Klasse von Änderungen betreffen den Inhalt von Cachelines. Bei der zweiten Klasse werden nur Attribute von Cachelines

geändert. Hierzu gehören Aktualisierungen der Zugriffszeit. Diese Änderungen werden durch Hits hervorgerufen. Wird auf eine Cacheline erfolgreich zugegriffen, das heißt gefunden, wird die Zeit des letzten Zugriffs aktualisiert. Dadurch ändert sich die Position der Cacheline innerhalb der Ordnung für die Ersetzung. Diese Änderung muss somit auch propagiert werden. In den nächsten Abschnitten wird die Propagation für beide Klassen im Detail betrachtet.

Inhaltliche Änderungen sind im Allgemeinen für die Funktion der PSMW von grundlegender Bedeutung. Ändert sich zum Beispiel die Menge der Knoten, die einen bestimmten Inhalt bereitstellen, hat diese Information starken Einfluss auf die Verfügbarkeit des Inhalts. Die Änderungen sollten also so schnell wie möglich verbreitet werden. Eine direkte Propagation (**direct Propagation**) bietet sich hier an. Die Änderung wird zum Zeitpunkt der Verarbeitung per CellBroadcast im Cluster propagiert. Jedes Mitglied des Clusters kann diese Änderung dann auch verarbeiten. Dabei ignorieren Knoten, die nicht für die entsprechende Restklasse zuständig sind, die Propagation. Knoten, die zuständig sind, verarbeiten die Propagation im lokalen Speicher. Änderungen, die per Propagation in einem Knoten durchgeführt werden, werden nicht erneut propagiert. Erstens ist dies innerhalb des Clusters bereits geschehen. Zweitens würden Propagation sonst in einer unendlich wachsenden Lawine weiter propagiert. Propagationen werden auch von Knoten ausgelöst, die nicht für die entsprechende Restklasse zuständig sind. Da Anwendungen immer nur mit der Cacheinstanz des lokalen Knotens arbeiten, würden Änderungen, für die eine Instanz nicht zuständig ist, verloren gehen. Jede Eintragung einer Cacheline hat somit eine Propagation dieser zur Folge. Die Replikate der Buckets werden dadurch clusterweit konsistent gehalten.

Die Änderung von Cachelineattributen kann gegenüber der inhaltlichen Änderung als geringer priorisiert betrachtet werden. Diese Änderungen beeinflussen die Handhabung von Cachelines, haben aber auf die Datenmächtigkeit des Caches keinen beziehungsweise wenig Einfluss. Diese Änderungen brauchen nicht direkt propagagiert werden. Sie werden in eine Warteschlange gesetzt. Propagationen dieser Art werden im Folgenden als **queued Propagation** bezeichnet. Ein CellBroadcastpaket mit einer direkten Propagation schöpft im Allgemeinen nicht die maximale Menge an Daten aus, die in einem Paket übertragen werden kann. Das Paket kann somit um queued Propagations erweitert werden. Es ist zwar unwahrscheinlich aber möglich, dass über einen längeren Zeitraum keine direkten Propagationen im Netz übertragen werden. Queued Propagations können dann auch nicht verbreitet werden. Ein weiteres Problem kann sich ergeben, wenn die Anzahl der queued Propagations so hoch ist, dass die Warteschlange nur durch direkte Propagationen nicht gelehrt wird. Die queued Propagations veralten dann in der Warteschlange. Sie werden nicht effektiv verbreitet. Um dies zu verhindern, wird periodisch die gesamte Warteschlange geleert, in dem alle queued Propagations, unabhängig von direkten Propagationen, versendet werden. Hierbei werden dann so viele Pakete versendet, wie nötig sind. Das Veralten von Propagationen ist damit ausgeschlossen. In Abbildung 3.5 ist das Handling der beiden Ereignisse dargestellt: links das Aktivitätsdiagramm für das Handling einer direkten Propagation, rechts das Aktivitätsdiagramm für das periodische Handling aller noch ausstehenden queued Propagations.

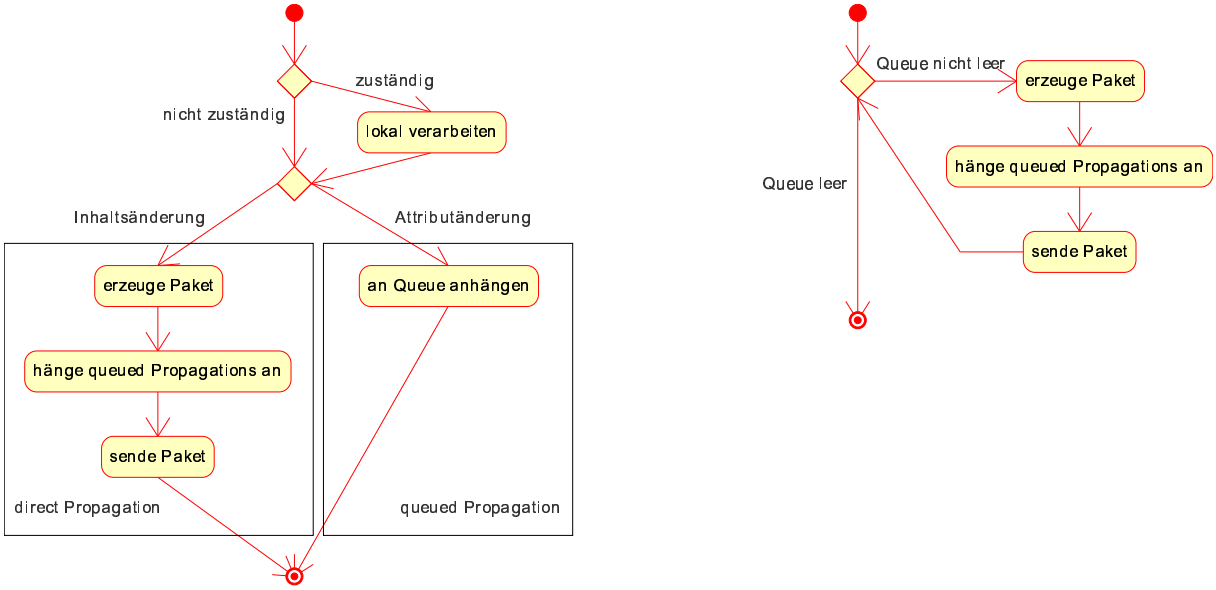


Abbildung 3.5: direct und periodicly queued Propagation

Empfängt ein Knoten ein Propagationspaket, entscheidet die Zuständigkeit des Knotens für die entsprechende Restklasse, ob das Paket weiter verarbeitet wird. Ist der Knoten für die Restklasse, zu der die propagierte Cacheline gehört, zuständig, wird diese im lokalen Cache aufgenommen beziehungsweise aktualisiert.

Für die Propagation über Clustergrenzen hinweg sind die Gateways zuständig. Gateways verarbeiten jede Propagation, die sie empfangen. Die Aufnahme beziehungsweise Aktualisierung im lokalen Cache des Gateways, ist wie bei anderen Knoten über die Zuständigkeit geregelt. Zusätzlich wird aber jede Propagation zur erneuten Propagation übernommen. Direkte Propagationen werden gleich erneut versendet, und queued Propagations werden in die Warteschlange des Gateways aufgenommen. Durch ein Flag im Propagationspaket werden die Propagationsklassen unterschieden. Das Flag wurde durch den Originator gesetzt. Um zu vermeiden, dass die Propagationen wieder in dem Cluster versendet werden, aus dem sie kommen, muss zu jeder empfangenen Propagation der Herkunftscluster gespeichert werden. Der Gatewayknoten ist dann in der Lage, CellBroadcastpakete über eine Clusteradressierung nur in den anderen Clustern zu versenden. Die Information, aus welchem Cluster eine Propagation empfangen wurde, reicht aber nicht aus, um Zyklen zu verhindern. Zum Beispiel ist es in einem Netz aus drei Clustern leicht möglich, einen Zyklus zu konstruieren. Der Gateway hat nur Wissen über den letzten Cluster, aber nicht vorherige Cluster. Wird von jedem Gateway die Information in welchem Cluster die Propagation bereits war, dem Paket zugefügt, entsteht so eine Sourceroute. Über diese Sourceroute können Gateways dann erfahren, ob die Propagation bereits in dem Cluster war. Ist dies der Fall, wird die Propagation ignoriert. Zyklen können so nicht mehr entstehen. Problematisch ist aber, dass Propagationen die maximale Größe eines Paketes ausnutzen. Die Aufzeichnung der Sourceroute benötigt aber weiteren Platz im Paket. Eine Sourceroute kann somit nicht als Lösung betrachtet werden. Eine zweite Möglichkeit ergibt sich durch die Verwendung von Sequenznummern. Um Propagationen im gesamten Netz eindeutig identifizieren zu können, ist eine Sequenznummer nicht ausreichend. Die Knoten haben Sequenzzähler, die unabhängig voneinander

arbeiten. Eine Synchronisation dieser Zähler kann als nicht realisierbar angesehen werden. Der dafür nötige Aufwand steht in keiner Relation zum Nutzen und widerspricht den Anforderungen, die für die Entwicklung des Systems aufgestellt wurden. Jeder Knoten kann für sich eine eindeutige Sequenznummer garantieren. Zusammen mit der Adresse der Knoten ergeben sich global eindeutige Sequenznummern. Zu jeder Propagation wird somit die Adresse des Originators und die von ihm vergebene Sequenznummer gespeichert und übertragen. Bei jeder Verarbeitung einer Propagation merken sich die Knoten die letzte Sequenznummer zu jedem Knoten. Wird nun bei jedem Empfang einer Propagation die Sequenznummer verglichen, werden Zyklen verhindert. Es werden immer nur Propagationen verarbeitet, deren Sequenznummer größer als die gespeicherte ist. Da jeder Knoten Gateway werden kann, wird die Speicherung der Sequenznummern von jedem Knoten durchgeführt. Selbst Clusterheads können Gateway werden, wenn sie ihre Aufgabe als Clusterhead verlieren. Das Merken des Clusters, über den eine Propagation empfangen wurde, fällt durch die Sequenznummern aber nicht weg. Um nicht unnütz Pakete in Clustern zu versenden, wird dieses Feature weiterhin benötigt. Abbildung 3.5a zeigt das resultierende Aktivitätsdiagramm für die Verarbeitung einer empfangenen Propagation.

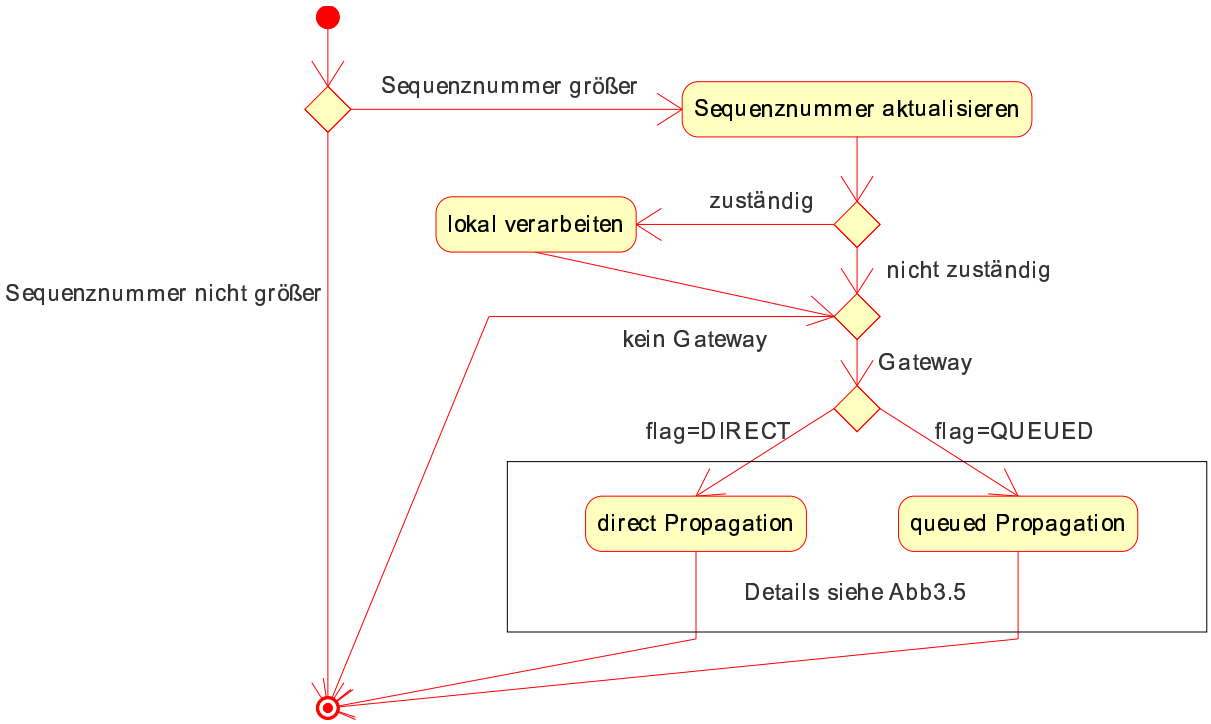


Abbildung 3.5a: Verarbeitung einer empfangenen Propagation

Die Propagationen verbreiten sich im gesamten Netzwerk Cluster für Cluster. Da direkte Propagationen gleich beim Empfang weitergeleitet werden, entspricht die Verweildauer innerhalb eines Cluster der Zeit, die benötigt wird, um eine direkte Propagation in einem Cluster zu verbreiten. Unter der optimistischen Annahme, dass dies innerhalb eines Round-Trips möglich ist, ergibt sich eine Verbreitungsgeschwindigkeit von einem Cluster pro Round-Trip. Bei einer Slotdauer von 10 Millisekunden und der Annahme von Clustern mit maximaler Grösse von 16 Mitgliedern ergibt sich folgende Formel:

$$\begin{aligned} v_{direct} &= \frac{1\,Cluster}{2\cdot(16-1)\cdot 0{,}01\,s} \\ &= 3\frac{1}{3}\,Cluster/s \end{aligned}$$

Neue und geänderte Cachelines verbreiten sich unter den gegebene Bedingungen somit innerhalb einer Sekunde in einem Radius von über 3 Clustern. Mit einer steigenden Übertragungsdauer innerhalb eines Clusters, die durch Echtzeitübertragungen hervorgerufen werden können, sinkt die Verbreitungsgeschwindigkeit indirekt proportional. Mit der Anzahl r an benötigten Round-Trips für die clusterinterne Propagation ändert sich die Formel folgendermaßen:

$$\begin{aligned} v_{direct}(r) &= \frac{1}{r}\cdot v_{direct} \\ &= 3\frac{1}{3r}\,Cluster/s \end{aligned}$$

Queued Propagations verbreiten sich weitaus gelassener. Außer der Verweildauer innerhalb eines Clusters ändert sich aber nichts an der Art, wie sie verbreitet werden. Die Verweildauer einzelner Propagationen ist davon abhängig, ob sie zur Füllung von Paketen mit direkten Propagationen genutzt werden. Ist dem so, ist die Verweildauer dieser Propagation zwischen der Dauer der Verbreitung innerhalb des Clusters und der maximalen Dauer zwischen den periodischen Propagationen. Diese Aussage trifft bei genauer Betrachtung der Verfahren auch auf queued Propagations zu, die nicht mit direkten Propagationen zusammen verbreitet werden. Trifft eine queued Propagation direkt vor der nächsten periodischen Propagation auf einem Gateway ein, ist ihre Verweildauer ebenfalls durch die Dauer der clusterinternen Propagation gekennzeichnet. Unter der Annahme, dass die queued Propagations ein Gateway gleichverteilt über die Propagationsperiode erreichen, kann somit für eine allgemeine Betrachtung von der halben Periodendauer ausgegangen werden. Unter denselben Annahmen, wie für die Verbreitungsgeschwindigkeit der direkten Propagationen $v_{direct}(r)$, und einer Periodendauer von 10 Sekunden ergeben sich folgende Formeln:

$$\begin{aligned} v_{queued} &= \frac{1\,Cluster}{2\cdot(16-1)\cdot 0{,}01\,s+\frac{1}{2}\cdot 10\,s} \\ &= \frac{10}{53}\,Cluster/s \end{aligned}$$

$$\begin{aligned} v_{queued}(r) &= \frac{1}{r}\cdot v_{queued} \\ &= \frac{10}{53r}\,Cluster/s \end{aligned}$$

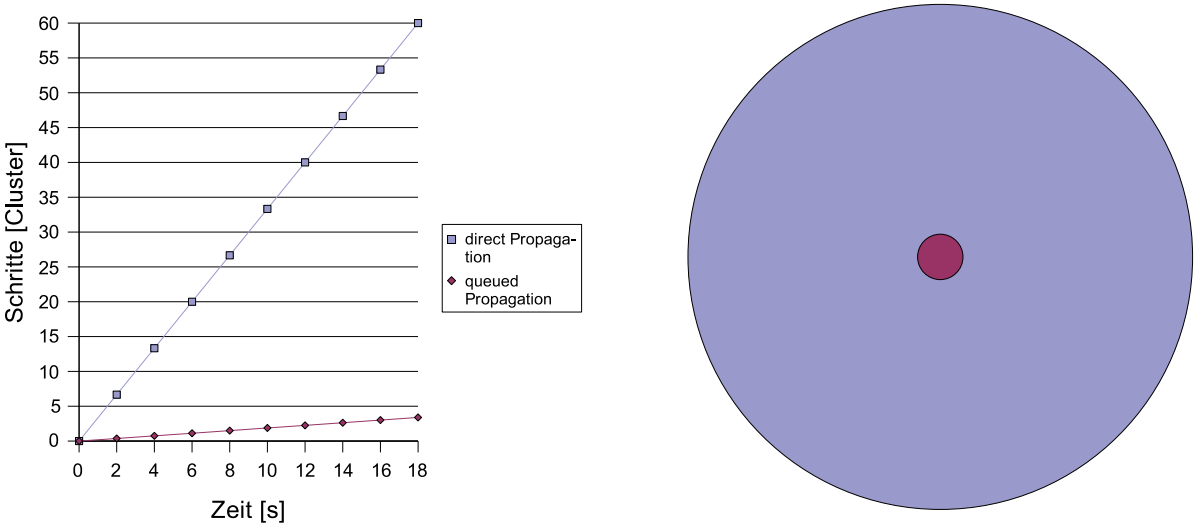


Abbildung 3.5b: Propagationsgeschwindigkeit

Abbildung 3.5b zeigt die resultierenden Verbreitungskurven für beide Propagationsarten links im Diagramm. Rechts ist die radiale Verbreitung beider Arten relativ zueinander dargestellt. Diese Verbreitung widerspiegelt ungefähr die Verbreitung in einem stark verbundenen Netzwerk.

Durch Übertragungsfehler werden die Propagationen gebremst. Eine lineare Verbreitung, wie hier dargestellt, ist unter realistischen Bedingungen nicht erreichbar. Stochastische Betrachtungen zur Propagation unter Einfluss von Übertragungsfehlern sind in Kapitel 3.7 zu finden.

3.6 Anfragen

Das Cachesystem ist netzwerktransparent. Applikationen, die es benutzten, haben kein Wissen über die Verteilung der Cachelines. Besteht der Cluster, in dem der Cache agiert, aus mehr als drei Knoten, werden die Cachelines disjunkt verteilt auf den Knoten gespeichert. Die lokalen Instanzen des Caches sind dann nicht für alle Cachelines zuständig. Wird eine Cacheline angefragt, für die die lokale Instanz nicht zuständig ist, müssen die andere Knoten des Clusters die Anfrage verarbeiten (Abbildung 3.6). Dazu wird ein CellBroadcast verwendet. Neben einem Flag zur Identifizierung als Anfrage wird der Schlüssel der gesuchten Cacheline übermittelt.

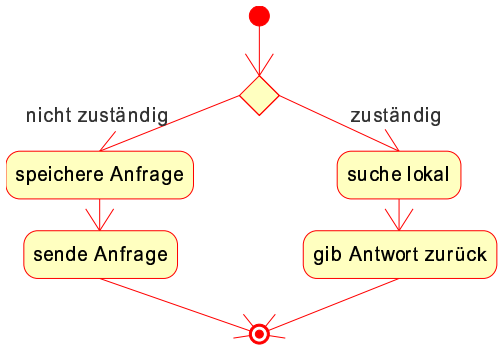


Abbildung 3.6: lokale Anfrage

Die Knoten des Clusters entscheiden dann analog zur lokalen Instanz anhand des Hashwertes des Schlüssels, ob die Anfrage weiter verarbeitet wird. Fällt die Entscheidung negativ aus, wird die Verarbeitung abgebrochen. Fällt sie dagegen positiv aus, wird nach der entsprechenden Cacheline gesucht. Kann keine Antwort gefunden werden, ist die Verarbeitung ebenfalls beendet. Eine negative Antwort auf die Anfrage wird nicht versendet, da sonst positive Antworten, die etwas später den anfragenden Knoten erreichen, nicht mehr an die Applikation hochgereicht werden können. Im anderen Fall eines positiven Resultats der Suche wird ein Antwortpaket per CellBroadcast gesendet (Abbildung 3.6a). Es enthält einen Flag zur Kennzeichnung als Antwortpaket und die vollständige Cacheline.

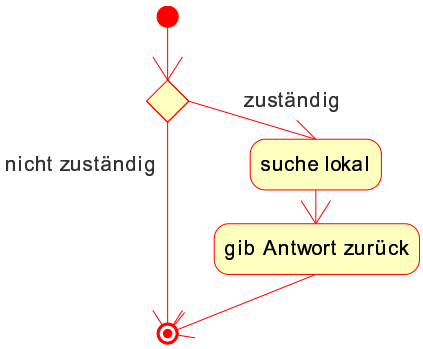


Abbildung 3.6a: Anfrage eines anderen Knotens

Die erste Antwort wird vom anfragenden Knoten übernommen und an die anfragende Applikation hochgereicht. Da es im Allgemeinen mehr als einen zuständigen Knoten gibt, werden auch mehrere Antwortpakete verschickt. Diese Pakete können nebenbei zur indirekten Teilsynchronisierung verwendet werden. Wird innerhalb eines Time-outs keine Antwort empfangen, reicht der Cache eine negative Antwort an die anfragende Applikation zurück (Abbildung 3.6b). Zum Hochreichen der Antworten werden, analog zur Verarbeitung von Ereignissen, Callbackfunktionen der anfragenden Applikation genutzt.

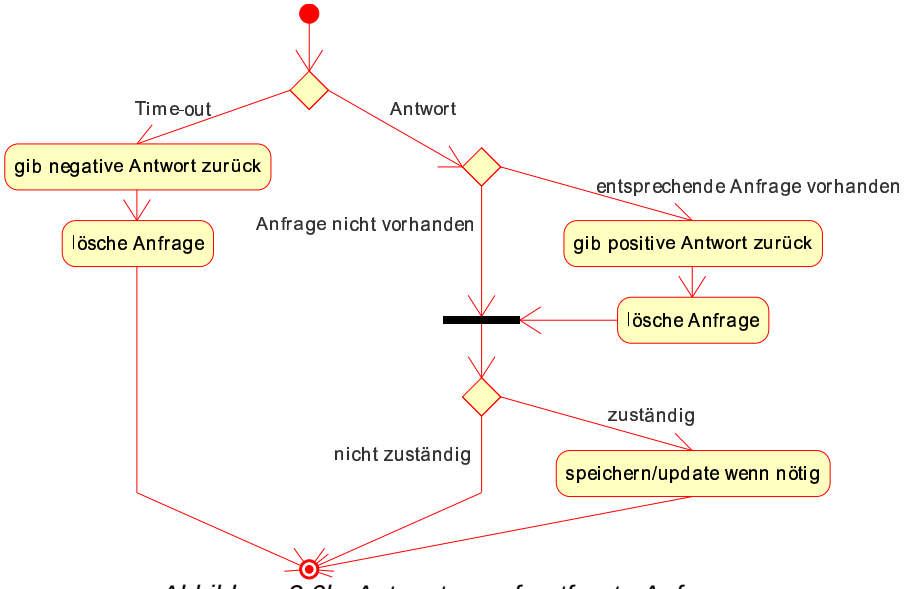


Abbildung 3.6b: Antworten auf entfernte Anfragen

Wird innerhalb eines definierten Time-outs keine entsprechende Antwort empfangen,

so wird eine negative Antwort zurückgegeben. Es gibt keine Rückmeldungen über negative Suchergebnisse der zuständigen Knoten. Ohne Time-out würde das System unendlich lange auf eine Antwort warten. Im Falle eines Time-outs obliegt es dann der Applikation, die gesuchte Information von der Quelle direkt zu ermitteln. Diese Information kann dann von der Applikation in den Cache eingetragen werden, damit sie beim nächsten mal schneller zur Verfügung steht.

Für die Beantwortung einer Suchanfrage wird unter idealen Bedingungen, dass heißt, wenn keine andere Kommunikation im Cluster stattfindet, minimal ein Round-Trip benötigt. Alle anderen Knoten müssen einmal in der Lage sein, eine Antwort zu senden. Unter realistischen Bedingungen können Antworten durchaus bis zu geschätzten 10 Round-Trips benötigen. Unter denselben Annahmen, wie in Kapitel 3.5, ergibt sich damit ein minimales Time-out von drei Sekunden. Eine entgültige Parametriesierung ist hier ebenfalls erst durch Tests realisierbar.

3.7 Stochastische Analyse der Kommunikation

In den vorherigen Kapiteln wurde die Funktionsweise des Cachesystems erläutert. In dem verteilten Konzept ist die Kommunikation die Basis des gesamten Systems. Durch die Beschränkung auf Best-Effort-Traffic der PSMW, und dort im speziellen dem CellBroadcast, ist die gesamte Kommunikation auf verbindungslose Übertragungen aufgebaut. Im CellBroadcast sind keine Empfangsbestätigungen vorhanden. Der Sender eines Paketes hat somit keine Rückmeldung über den Erfolg des Empfangs. Unter der Annahme, dass keine Fehler bei der Übertragung von Paketen entstehen, ist das kein Problem, da der Erfolg dann immer garantiert ist. In realen Umgebungen von Funknetzwerken existiert bei der Übertragung immer eine Fehlerrate. Abhängig von dem Funknetzwerk und der momentanen Umgebung in der es sich befindet variiert die Fehlerrate. Die Kombination der verbindungslosen Kommunikation des Cachesystems und der Existenz von Übertragungsfehlern erlaubt somit nur eine Aussage über die Wahrscheinlichkeit, mit der das System erfolgreich kommuniziert. In diesem Kapitel wird das System stochastisch analysiert, um genau solche Aussagen über die Zuverlässigkeit der Kommunikation des Cachesystems zu machen.

Die PSMW wird im WLAN (IEEE802.11) eingesetzt. Experimente der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ haben einen durchschnittlichen Übertragungsfehler von 3% bis 10% ergeben. Das Umfeld der Tests waren IEEE802.11[a|b|g] Netze unter verschiedenen realitätsnahen Bedingungen. Die folgenden Betrachtungen basieren auf diesem Erfahrungswert.

3.7.1 Basiskommunikation

Um die Kommunikationsprozesse zu analysieren, werden zuerst einzelne Basisübertragungen analysiert. Aus diesen Zwischenergebnissen lassen sich dann die Analysen der einzelnen Kommunikationsprozesse ableiten und zusammensetzen.

Das Ereignis, dass eine Datenübertragung unidirektional von einem zum anderen Knoten ihr Ziel erfolgreich erreicht, ist in den folgenden Betrachtungen A.Die Wahrscheinlichkeit, dass dieses Ereignis eintrifft, entspricht dem Komplement der Wahrscheinlichkeit eines Übertragungsfehlers. Mit dem Ereignis E als Eintreffen eines Fehlers und der zugehörigen Wahrscheinlichkeit $P(E)=\epsilon \approx 0{,}03..0{,}1$, ergibt sich folgende Wahrscheinlichkeit für eine erfolgreiche unidirektionale Punkt-zu-Punkt-Übertragung:

$$\begin{aligned} A &: \textit{erfolgreiche Punkt}-\textit{zu}-\textit{Punkt}-\textit{Übertragung} \\ P(A,\epsilon) &= 1-P(E) \\ &= 1-\epsilon \end{aligned}$$

Für den Erfolg verschiedener Vorgänge im Cache sind auf Grund der Redundanz und der Fehlertoleranz des Systems weniger Punkt-zu-Punkt-Übertragungen als vielmehr Punkt-zu-Multipunkt-Übertragungen verantwortlich. Jede Übertragung eines Paketes wird vom Clusterhead per CellBroadcast an alle Clients eines Clusters gleichzeitg ausgeführt. Der Empfang dieser Übertragungen kann bei den Clients abhängig wie auch unabhängig voneinander sein. In einem drahtlosen Netzwerk können sehr unterschiedliche Störungen auftreten. Die Spanne reicht von Störungen einzelner Knoten bis hin zu globalen Störungen. Eine Aussage über die Abhängigkeit des Empfangs eines Paketes bei den einzelnen Knoten kann somit nicht gemacht werden. Aus diesem Grund werden in den folgenden Analysen immer beide Extremfälle betrachtet. Die resultierenden Wahrscheinlichkeiten sind dabei immer mit *global* und *local* gekennzeichnet.

Das Ereignis einer erfolgreichen unidirektionalen Punkt-zu-Multipunkt-Übertragung eines eizelnen Paketes ist im Folgenden B. Die Wahrscheinlichkeit für B_{global} ergibt sich direkt aus der Wahrscheinlichkeit von A. Die Empfänger sind hierbei nicht unabhängig gegenüber dem Empfang des Paketes. Das Ereignis B_{global} ist somit äquivalent zum Ereignis A.

$$\begin{aligned} B_{global} &: \begin{array}{c}\textit{erfolgreiche Punkt}-\textit{zu}-\textit{Multipunkt}-\textit{Übertragung} \\ \textit{bei globalen Störquellen}\end{array} \\ P(B_{global},\epsilon) &= P(A,\epsilon) \\ &= 1-\epsilon \end{aligned}$$

Unter der Annahme von lokalen Störquellen ist es möglich, dass eine Teilmenge der Empfänger erfolgreich empfangen hat, wogegen die Komplementärmenge nicht erfolgreich empfangen hat. Der Empfang der einzelnen Knoten ist nicht voneinander abhängig. Die Wahrscheinlichkeit des Ereignisses B_{local} ist somit von der Verteilung der einzelnen Empfangereignisse abhängig. Die Verteilung der Empfangsereignisse ist somit binomialverteilt, da die Einzelereignisse nicht voneinander abhängig sind, und immer ein Teil dieser diskreten Menge von Ereignissen eintritt. Die Teilereignisse entsprechen dabei dem Ereignis A. Die Punkt-zu-Multipunkt-Übertragung wird hierbei als eine Menge von unabhängigen Punkt-zu-Punkt-Übertragungen modelliert.

$$\begin{aligned} B_{local} &: \begin{array}{c}\textit{erfolgreiche Punkt}-\textit{zu}-\textit{Multipunkt}-\textit{Übertragung} \\ \textit{bei lokalen Störquellen}\end{array} \\ n &: \textit{Anzahl Empfänger} \\ r &: \textit{Anzahl Empfänger , die empfangen} \end{aligned}$$

$$\begin{aligned} P(B_{local},\epsilon,n,r) &= Bio(n,r,P(A,\epsilon)) \\ &= \binom{n}{r}(1-\epsilon)^r \epsilon^{n-r} \qquad r \in \mathbb{N} \wedge 0 \le r \le n \end{aligned}$$

Die Kommunikationsbasis des Cachesystem ist der CellBroadcast in einem Cluster. Wie in Kapitel 2.7 beschrieben, wird dabei das Paket vom Client zum Clusterhead übertragen, der es dann versendet. Der CellBroadcast besteht somit aus zwei Teilübertragungen. Die Übertragung des Paketes vom Client zum Clusterhead entspricht dem Ereignis A, einer Punkt-zu-Punkt-Übertagung. Von einem Knoten muss genau zu einem anderen Knoten, dem Clusterhead, übertragen werden. Der zweite Teil wird durch das Ereignis B, einer Punkt zu Multipunkt Übertragung, beschrieben. Von einem Knoten, dem Clusterhead, wird zu einer Menge von anderen Knoten übertragen. Eine Ausnahme ergibt sich, wenn der Clusterhead selbst ein Paket per CellBroadcast versendet. Dann fällt der erste Schritt weg. Für die Betrachtung des Ereignisses C, dass ein Paket innerhalb eines Clusters erfolgreich übertragen wird, müssen die zwei Fälle beachtet werden. Für die Wahrscheinlichkeit eines erfolgreichen CellBroadcasts ergibt sich somit folgende Formel:

$$\begin{aligned} C &: \textit{erfolgreicher CellBroadcast} \\ n &: \textit{Anzahl Clients} \\ r &: \textit{Anzahl Clients, die empfangen} \\ P(C,\epsilon,n,r) &= \begin{cases} P(B,\epsilon,n,r) & \textit{Sender ist Clusterhead} \\ P(A \cap B,\epsilon,n,r) & \textit{Sender ist Client} \end{cases} \end{aligned}$$

Um nicht immer die zwei Fälle, ob der Sender der Clusterhead ist oder nicht, zu betrachten, werden beide Terme über die Wahrscheinlichkeit, mir der der Sender der Clusterhead ist, zusammengeführt. Die Wahrscheinlichkeit, mit der ein bestimmter Knoten der Sender eines Paketes ist, ist gleichverteilt. Dass der Sender der Clusterhead ist, tritt genau einmal auf. Alle anderen Sender sind dann Clients. Da n die Anzahl der Clients ist, ist $n+1$ die Anzahl der Mitglieder, und somit die Anzahl der möglichen Sender. Die Wahrscheinlichkeit für einen erfolgreichen CellBroadcast sieht dann folgendermaßen aus:

$$P(C,\epsilon,n,r) = \frac{1}{n+1}(P(B,\epsilon,n,r)+n \cdot P(A \cap B,\epsilon,n,r))$$

Die zweigeteilte Betrachtung durch lokale und globale Störquellen bei dem Ereignis B muss hier weitergeführt werden.

$$\begin{aligned} C_{global} &: \textit{erfolgreicher CellBroadcast bei globalen Störquellen} \\ P(C_{global},\epsilon) &= \frac{1}{n+1}(P(B_{global},\epsilon)+n \cdot P(A \cap B_{global},\epsilon)) \\ &= \frac{1}{n+1}((1-\epsilon)+n \cdot (1-\epsilon)^2) \\ &= \frac{1}{n+1}(1-\epsilon+n \cdot (1-2\epsilon+\epsilon^2)) \\ &= \frac{1}{n+1}(1+n+n\epsilon^2-\epsilon-2n\epsilon) \end{aligned}$$

$$
\begin{aligned}
C_{local} \quad &: \quad \textit{erfolgreicher CellBroadcast bei lokalen Störquellen} \\
P(C_{local},\epsilon,n,r) &= \frac{1}{n+1}\left(P(B_{local},\epsilon,n,r)+n\cdot P(A\cap B_{local},\epsilon,n,r)\right) \\
&= \frac{1}{n+1}\left(\binom{n}{r}(1-\epsilon)^r\epsilon^{n-r}+n\cdot(1-\epsilon)\cdot\binom{n}{r}(1-\epsilon)^r\epsilon^{n-r}\right) \\
&= \left(1-\frac{n\epsilon}{n+1}\right)\binom{n}{r}(1-\epsilon)^r\epsilon^{n-r}
\end{aligned}
$$

3.7.2 Kommunikationsprozesse des Cachesystems

Der erste Kommunikationsprozess, der hier betrachtet wird, ist die **Konfiguration** des Cachesystems in einem Cluster. In Kapitel 3.1.4 wurde das Push-Verfahren mit Wiederholungen zur Konfiguration als beste Lösung ermittelt. Ohne Wiederholungen des Paketes ist die erfolgreiche Konfiguration davon abhängig, dass ein CellBroadcast alle Clients empfängt. Da der Sender der Konfguration immer der Clusterhead ist, entspricht das Ereignis D, der erfolgreichen Konfigurationsübertragung, dem Ereignis B, der erfolgreichen Punkt-zu-Multipunkt-Übertragung. Dabei müssen hier alle Clients das Paket empfangen. Die Anzahl der Wiederholungen wurde an der Anzahl der Clients festgemacht. Die Wiederholungen des Konfigurationspaketes können als unabhängig voneinander betrachtet werden, da sie zu unterschiedlichen Zeitpunkten im Netz übertragen werden. Die Verteilung der Wahrscheinlichkeiten der diskreten, unabhängigen Einzelereignisse ist binomialverteilt. Die Übertragung ist erfolgreich, wenn mindestens eine Einzelübertragung erfolgreich ist. Die Wahrscheinlichkeit des Ereignisses D lässt sich dann wie folgt berechnen:

$$
\begin{aligned}
D \quad &: \quad \textit{erfolgreiche Konfigurationsübertragung} \\
P(D,\epsilon,n) &= 1-\binom{n}{0}P(B,\epsilon,n,r=n)^0(1-P(B,\epsilon,n,r=n))^n \\
&= 1-(1-P(B,\epsilon,n,r=n))^n
\end{aligned}
$$

Die beiden Arten der Störquellen müssen hier wieder beachtet werden. Es entstehen zwei Formeln. Eine in Abhängigkeit von B_{global} und eine in Abhängigkeit von B_{local}.

$$
\begin{aligned}
D_{global} \quad &: \quad \textit{erfolgreiche Konfigurationsübertragung bei globalen Störquellen} \\
P(D_{global},\epsilon,n) &= 1-(1-P(B_{global},\epsilon))^n \\
&= 1-(1-(1-\epsilon))^n \\
&= 1-\epsilon^n
\end{aligned}
$$

$$
\begin{aligned}
D_{local} \quad &: \quad \textit{erfolgreiche Konfigurationsübertragung bei lokalen Störquellen} \\
P(D_{local},\epsilon,n) &= 1-(1-P(B_{local},\epsilon,n,r=n))^n \\
&= 1-\left(1-\left(\binom{n}{n}(1-\epsilon)^n\epsilon^0\right)\right)^n
\end{aligned}
$$

$$= \quad 1-(1-(1-\epsilon)^n)^n$$

Für die grafische Auswertung dieser Formeln werden zwei Cluster mit fünf und zehn Mitgliedern betrachtet. Mit einem Mitglied als Clusterhead haben die Cluster somit vier und neun Clients.

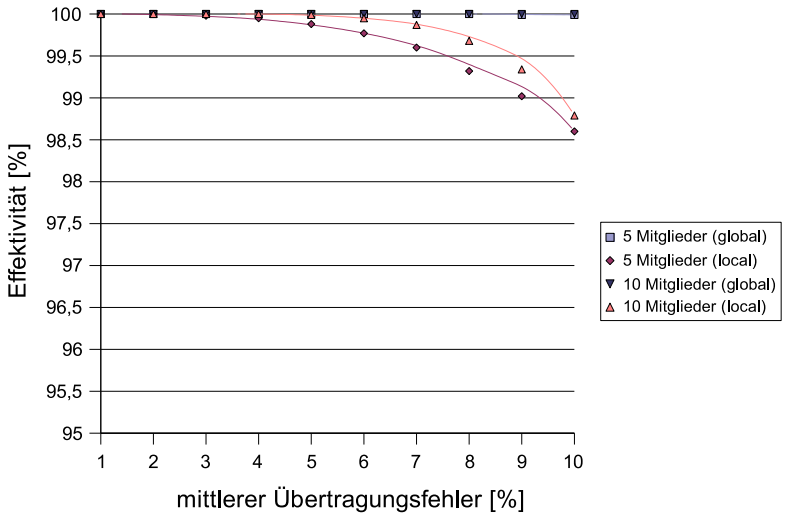


Abbildung 3.7.2: Konfiguration

Die sehr hohen Werte der Übertragungseffektivität erklären sich durch die vielen Wiederholungen. Da die Größe eines Clusters auf 16 Mitglieder beschränkt ist, stellt die häufige Wiederholung auch kein Problem dar. Sie wird somit maximal 15 Mal versendet. Selbst bei einer durchschnittlichen Fehlerrate von 10% wird die Konfiguration noch mit über 98 prozentiger Wahrscheinlichkeit an alle Clients übertragen. Die relativ hohe Anforderung an die Zuverlässigkeit der Konfigurationsübertragungen, wie sie in Kapitel 3.1.4 gefordert wird, ist somit erfüllt.

Als Nächstes wird der komplexeste Kommunikationsprozess des Systems untersucht. Die **Synchronisation** besteht aus einer Menge voneinander abhängiger Kommunikationen, die alle erfolgreich sein müssen, damit die gesamte Synchronisation erfolgreich ist. Als Erstes muss die Anfrage versendet werden. Dabei handelt es sich um einen CellBroadcast, der von mindestens einem zuständigen Mitglied empfangen werden muss. Die Anzahl der zuständigen Mitglieder m ist durch $m=\sqrt{n+1}$ gegeben. Unter der Annahme, dass einer dieser zuständigen Mitglieder gerade synchronisieren muss, ergibt sich $m=\sqrt{n+1}-1$.

$$\begin{aligned}
F_1 \quad &: \quad \textit{erfolgreiche Synchronisationsanfrage} \\
m \quad &: \quad \textit{Anzahl zuständiger Knoten} \\
P(F_1, \epsilon, m) \quad &= \quad P(C, \epsilon, n=m, r \geq 1) \\
&= \quad 1-P(C, \epsilon, n=m, r=0) \\
&= \quad 1- \\
&= \quad \frac{1}{m+1}\left(P(B, \epsilon, n=m, r=0)+m \cdot P(A \cap B, \epsilon, n=m, r=0)\right) \\
&= \quad 1-\frac{1}{m+1}P(B, \epsilon, n=m, r=0)(1+m \cdot P(A, \epsilon))
\end{aligned}$$

$$= 1-\frac{1}{m+1}P(B,\epsilon,n=m,r=0)(m+1-m\epsilon)$$
$$= 1-(1-\frac{m\epsilon}{m+1})P(B,\epsilon,n=m,r=0)$$

Der zweite Schritt ist die Übertragung der Synchronisationsbereitschaft. Es handelt sich hier um eine Menge von Punkt-zu-Punkt-Übertragungen. Eine Menge von potentiellen Synchronisationssendern versenden die Bereitschaft. Ziel ist hier dann der einzelne Synchronisationsempfänger. Die Einzelübertragungen sind unabhängig voneinander. Demzufolge sind sie binomialverteilt. Für den Erfolg der Bereitschaftsübertragung muss nur eine Bereitschaft erfolgreich versendet werden.

$$F_2 : \begin{array}{l}\textit{erfolgreiche Übertragung}\\ \textit{der Synchronisationsbereitschaft}\end{array}$$
$$P(F_{2,}\epsilon,m) = Bio(m,r\geq 1,P(A,\epsilon))$$
$$= 1-\binom{m}{0}P(A,\epsilon)^0(1-P(A,\epsilon))^m$$
$$= 1-\epsilon^m$$

Nachdem die Bereitschaft erfolgreich empfangen wurde, sendet der Synchronisationsempfänger ein Startpaket für einen potentiellen Synchronisationssender. Die Übertragung entspricht der Punkt-zu-Punkt-Übertragung.

$$F_3 : \textit{erfolgreiche Übertragung des Startpaketes}$$
$$P(F_{3,}\epsilon) = P(A,\epsilon)$$
$$= 1-\epsilon$$

Die Übertragung der Daten wird vom Sender nach Empfang des Startpaketes begonnen. Im Allgemeinen handelt es sich hier, durch die Menge zu übertragender Daten, um eine Menge von Paketen. Die Wahrscheinlichkeit, wieviele Pakete davon übertragen werden, ergibt sich direkt aus der Wahrscheinlichkeit $P(A,\epsilon)$. In Kapitel 3.4 wurde bereits eine akzeptierte Fehlerrate von 10% verlorener Datenpakete eingeführt. Die Übertragung der Daten ist somit genau dann erfolgreich wenn mindestens 90% der Pakete empfangen werden. Unter der Annahme von bis zu 10% durchschnittlicher Fehlerrate, ist die Übertragung der Synchronisationsdaten immer erfolgreich.

$$F_4 : \begin{array}{l}\textit{erfolgreiche Übertragung}\\ \textit{der Synchronisationsdaten}\end{array}$$
$$P(F_{4,}\epsilon) = \begin{cases}1 & \epsilon\leq 0{,}1\\ 0 & \epsilon>0{,}1\end{cases}$$

Nach dem letzten Datenpaket sendet der Synchronisationssender ein Schlusspaket. Diese Übertragung entspricht, wie die Übertragung des Startpaketes, einer Punkt-zu-Punkt-Übertragung.

$$F_5 : \textit{erfolgreiche Übertragung des Schlusspaketes}$$
$$P(F_{5,}\epsilon) = P(A,\epsilon)$$
$$= 1-\epsilon$$

Alle fünf Teilereignisse sind voneinander abhängig. Die Wahrscheinlichkeit für eine erfolgreiche Synchronisation, Ereignis F, bildet sich durch die Kombination der Teilereignisse. Für die Übertragung der Datenpakete wird angenommen, dass die durchschnittliche Fehlerrate des Netzwerkes 10% nicht übersteigt. Anderen falls wird bei einer Wiederhohlung der Synchronisation die Grenze der akzeptierten Fehlerrate nach oben korrigiert. Bei Fehlerraten des Netzwerkes von über 10% werden starke Inkonsistenzen der Datenbestände in Kauf genommen, da eine Synchronisierung dann keinen großen Erfolg mehr haben kann. Die Betrachtung der Synchronisation ist somit nur bis zu einer Fehlerrate von 10% sinnvoll.

$$
\begin{aligned}
F &: \textit{erfolgreiche Synchronisation} \qquad \epsilon \le 0{,}1 \\
P(F,\epsilon,m) &= P(F_1 \cap F_2 \cap F_3 \cap F_4 \cap F_5,\epsilon,m) \\
&= P(F_1,\epsilon,m)\cdot P(F_2,\epsilon,m)\cdot P(F_3,\epsilon)\cdot P(F_4,\epsilon)\cdot P(F_5,\epsilon) \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)P(B,\epsilon,n=m,r=0)\right)(1-\epsilon^m)(1-\epsilon)^2
\end{aligned}
$$

In der Formel ist das Ereignis B noch nicht aufgelöst, da hier wieder die zwei Fälle von Störquellen beachtet werden müssen. Das Ereignis B kommt nur bei der Übertragung der Anfrage zum Tragen. Alle folgenden Teilübertragungen beziehen sich auf Punkt-zu-Punkt-Übertragungen, wo beide Arten von Störungen den selben Einfluss haben. Zu beachten ist dabei, das $P(B_{global},\epsilon,n,r=0)$ den Fall darstellt, bei dem kein Empfänger die Übertragung empfängt. Bei globalen Störquellen folgt daraus $P(B_{global},\epsilon,n,r=0)=1-P(B_{global},\epsilon)$. Es ergeben sich folgende zwei Formeln:

$$
\begin{aligned}
F_{global} &: \textit{erfolgreiche Synchronisation bei globalen Störquellen} \\
P(F_{global},\epsilon,m) &= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)(1-P(B_{global},\epsilon))\right)(1-\epsilon^m)(1-\epsilon)^2 \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)\epsilon\right)(1-\epsilon^m)(1-\epsilon)^2 \\
&= \frac{1}{m+1}(1+m+m\epsilon^2-(m+1)\epsilon)(1-\epsilon^m)(1-\epsilon)^2 \\
F_{local} &: \textit{erfolgreiche Synchronisation bei lokalen Störquellen} \\
P(F_{local},\epsilon,m) &= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)P(B_{local},\epsilon,n=m,r=0)\right)(1-\epsilon^m)(1-\epsilon)^2 \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)\left(\binom{m}{0}(1-\epsilon)^0\epsilon^m\right)\right)(1-\epsilon^m)(1-\epsilon)^2 \\
&= \frac{1}{m+1}(1+m+m\epsilon^{m+1}-(m+1)\epsilon^m)(1-\epsilon^m)(1-\epsilon)^2
\end{aligned}
$$

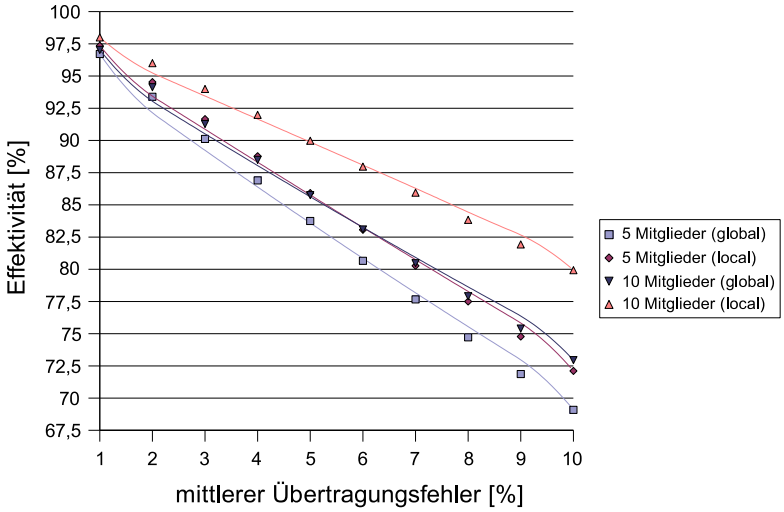


Abbildung 3.7.2a: Synchronisierung

In Abbildung 3.7.2a sind die Ergebnisse für beide Beispielcluster mit fünf und zehn Knoten dargestellt. In Clustern mit mehr Mitgliedern steigt die Effektivität der Kommunikation des Synchronisationsprozesse. Bei der Annahme von lokalen Störquellen ergeben sich nochmals höherer Ergebnisse. Beides ist auf die Punkt-zu-Multipunkt-Übertragung der Anfrage zurückzuführen. Nur dort hat die Anzahl der Mitglieder und die Art der Störquellen einen Einfluss. Auf die anderen Teilübertragungen gibt es keinen Einfluss. Deshalb sind die Unterschiede der Ergebnisse auch nicht sonderlich groß.

Um die Effektivität, die in Abbildung 3.7.2a doch recht weit unter 90% geht, zu erhöhen, können Übertragungswiederholungen eingeführt werden. In Kapitel 3.4 ist das bereits erwähnt. Am empfindlichsten reagieren die beiden Teilübertragungen des Startpaketes und des Schlusspaketes auf eine Erhöhung des mittleren Übertragungsfehlers des Netzwerkes. Es sind jeweils nur einzelne Punkt-zu-Punkt-Übertragungen, die durch eine Wahrscheinlichkeit von $1-\epsilon$ definiert sind. Um den Effekt der Übertragungwiederhohlungen zu verdeutlichen, wird als Nächstes die Synchronisation noch einmal mit betrachtet. Dabei werden aber das Start- und das Schlusspaket doppelt übertragen. Beide Übertragungen sind dann binomialverteilt über zwei Versuche.

$$
\begin{aligned}
F_3' &: \textit{doppelte Übertragung des Startpaketes} \\
P(F_3',\epsilon) &= Bio(2, r \geq 1, P(A,\epsilon)) \\
&= 1 - Bio(2, r = 0, P(A,\epsilon)) \\
&= 1 - \binom{2}{0} P(A,\epsilon)^0 (1 - P(A,\epsilon))^2 \\
&= 1 - \epsilon^2
\end{aligned}
$$

$$
\begin{aligned}
F_5' &: \textit{doppelte Übertragung des Schlusspaketes} \\
P(F_5',\epsilon) &= Bio(2, r \geq 1, P(A,\epsilon)) \\
&= 1 - \epsilon^2
\end{aligned}
$$

$$
F' : \textit{geänderte Synchronisation} \qquad \epsilon \leq 0{,}1
$$

$$P(F',\epsilon,m) = \left(1-\left(1-\frac{m\epsilon}{m+1}\right)P(B,\epsilon,n=m,r=0)\right)(1-\epsilon^m)(1-\epsilon^2)^2$$

$$F_{global}' : \textit{geänderte Synchronisation bei globalen Störquellen}$$

$$P(F_{global}',\epsilon,m) = \frac{1}{m+1}(1+m+m\epsilon^2-(m+1)\epsilon)(1-\epsilon^m)(1-\epsilon^2)^2$$

$$F_{local}' : \textit{geänderte Synchronisation bei lokalen Störquellen}$$

$$P(F_{local}',\epsilon,m) = \frac{1}{m+1}(1+m+m\epsilon^{m+1}-(m+1)\epsilon^m)(1-\epsilon^m)(1-\epsilon^2)^2$$

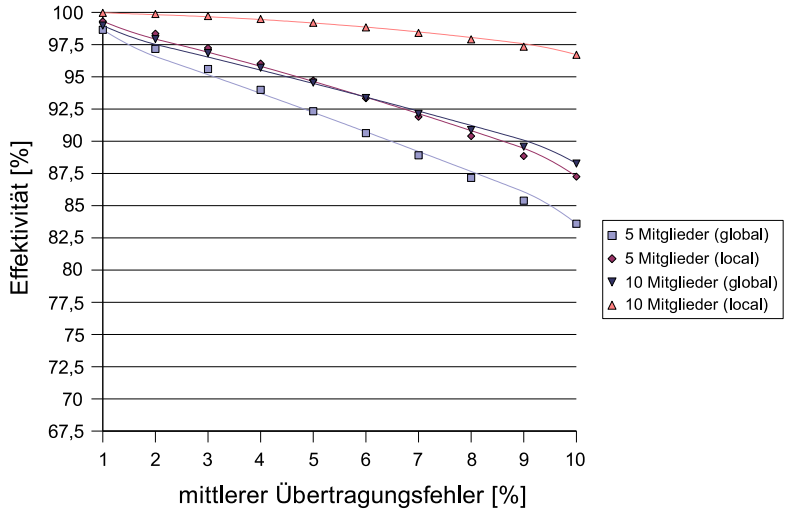


Abbildung 3.7.2b: Synchronisiation mit doppelten Start- und Schlusspaketen

Abbildung 3.7.2b zeigt, dass bereits das doppelte Senden des Start- und des Schlusspaketes eine starke Verbesserung herbeiführt. Bis zu einer durchschnittlichen Rate des Übertragungsfehlers von 6% geht die Effektivität der Synchronisationskommunikation nicht unter 90%. Diese Effektivität kann als akzeptabel angesehen werden.

Der nächste zu betrachtende Kommunikationsprozess ist die **Propagation**. Sie wird in zwei Schritten betrachtet. Als Erstes wird nur die clusterinterne Propagation betrachtet. Diese ist für alle Eintragungen und Änderungen von Cachelines zuständig. Eine Propagation wird hierbei mit Hilfe eines CellBroadcastes im Cluster verbreitet. Da immer mehr als ein Mitglied für die Verarbeitung einer Cacheline zuständig ist, handelt es sich um eine Punkt-zu-Multipunkt-Übertragung. Eine Propagation kann als erfolgreich angenommen werden, wenn nur ein Zielknoten diese empfängt. Beim Eintragen einer neuen Cacheline, ist diese dann zwar nur auf einem Replikat gespeichert, wird aber auf sie zugegriffen, ändert sich ihre Zugriffszeit. Dies hat zur Folge, dass sie erneut propagaiert wird, wodurch die Chance, dass sie auf den anderen Replikaten auch gespeichert wird, wieder erhöht wird. Die Beschränkung auf mindestens einen erfolgreichen Empfang ist damit ausreichend.

$$G : \textit{erfolgreiche clusterinterne Propagation}$$

$$m : \textit{Anzahl zuständiger Knoten} \qquad m=\sqrt{n+1}$$

$$\begin{aligned} P(G,\epsilon,m) &= P(C,\epsilon,n=m,r\geq 1) \\ &= 1-P(C,\epsilon,n=m,r=0) \qquad \textit{siehe } F_1 \\ &= 1-\left(1-\frac{m\epsilon}{m+1}\right)P(B,\epsilon,n=m,r=0) \end{aligned}$$

Da es sich um eine Punkt-zu-Mulitpunkt-Übertragung handelt, müssen die zwei unterschiedlichen Arten von Störquellen wieder extra betrachtet werden.

$$G_{global} : \textit{erfolgreiche clusterinterne Propagation bei globalen Störquellen}$$

$$\begin{aligned} P(G_{global},\epsilon,m) &= 1-\left(1-\frac{m\epsilon}{m+1}\right)P(B_{global},\epsilon,n=m,r=0) \\ &= 1-\left(1-\frac{m\epsilon}{m+1}\right)(1-P(B_{global},\epsilon)) \\ &= 1-\left(1-\frac{m\epsilon}{m+1}\right)\epsilon \\ &= \frac{1}{m+1}(1+m+m\epsilon^2-(m+1)\epsilon) \end{aligned}$$

$$G_{local} : \textit{erfolgreiche clusterinterne Propagation bei lokalen Störquellen}$$

$$\begin{aligned} P(G_{local},\epsilon,m) &= 1-\left(1-\frac{m\epsilon}{m+1}\right)P(B_{local},\epsilon,n=m,r=0) \\ &= 1-\left(1-\frac{m\epsilon}{m+1}\right)\binom{m}{0}(1-\epsilon)^0\epsilon^m \\ &= 1-\left(1-\frac{m\epsilon}{m+1}\right)\epsilon^m \\ &= \frac{1}{m+1}(1+m+m\epsilon^{m+2}-(m+1)\epsilon^m) \end{aligned}$$

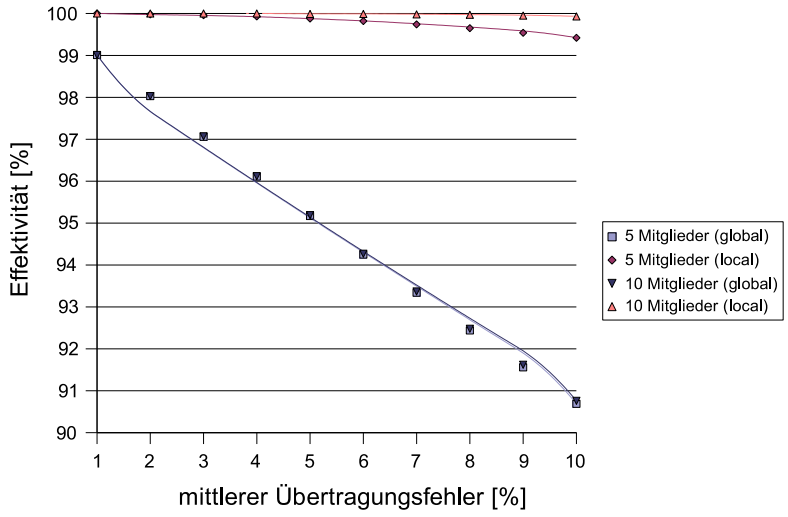


Abbildung 3.7.2c: clusterinterne Propagation

Die Ergebnisse für die Propagation in Abbildung 3.7.2c sind durchweg über 90%. Es

wird somit eine recht hohe Effektivität der Propagationskommunikation erzielt. Eine deutliche Differenz ist zwischen den Kurven der globalen und lokalen Störquellen zu sehen. Verantwortlich dafür ist der Fakt, dass angenommen wurde, dass eine Propagation mit einem Empfang bereits erfolgreich ist. Unter der Annahme von globalen Störquellen hat diese hinreichende Forderung keinen Einfluss, denn hier empfangen entweder alle oder keine Zielknoten. Wird angenommen, dass eine Propagation von allen Zielknoten empfangen werden muss, ist zu erwarten, dass die Ergebnisse beider Betrachtungen identisch sind.

Die zweite Möglichkeit der Propagation ist die **clusterübergreidende Propagation**. Sie besteht aus einer Kaskadierung von clusterinternen Propagationen. Da nur die Gateways für die Verbreitung in andere Cluster zuständig sind, ist bei der Übertragung im Cluster wichtig, dass diese das Paket empfangen. Die Anzahl der Gateways ist aber in keiner Hinsicht definiert. Es ist möglich, dass gar kein Gateway existiert. Es ist aber auch möglich, dass alle Clients Gateways sind. Die Wahrscheinlichkeit des Ereignisses H_1, dass eine Propagation zu einem angrenzenden Cluster weitergeleitet wird, ist somit von der Gatewayanzahl g abhängig. H_1 tritt genau dann ein, wenn mindestens ein Gateway das Paket empfängt, weil die Propagation dann kaskadieren kann. Damit eine Propagation durch einen Cluster weitergeleitet werden kann, sind mindestens zwei Gateways notwendig. Das Gateway, über den die Propagation einen Cluster erreicht, kann diese nicht weiterleiten. In der Formel muss die Anzahl der potentiellen Empfänger mit $g-1$ angenommen werden.

$$
\begin{aligned}
H_1 &: \textit{Propagation zu angrenzendem Cluster} \\
P(H_{1,}\epsilon, g) &= P(C, n=g-1, r\geq 1) \\
&= 1-P(C, n=g-1, r=0) \\
&= \frac{1}{g}(g-(2+g-(g+1)\epsilon)P(B,\epsilon, n=g-1, r=0))
\end{aligned}
$$

Auf einem einzelnen Pfad der Propagation betrachtet, sind die Übertragungen von Cluster zu Cluster voneinander abhängige Ereignisse. Mit steigender Entfernung wird die Wahrscheinlichkeit der Propagation sinken. Die Berechnung ist damit von der Anzahl der Propagationsschritte p abhängig. Die Wahrscheinlichkeit der clusterübergreifenden Propagation lässt sich dann wie folgt modellieren.

$$
\begin{aligned}
H &: \textit{clusterübergreifende Propagation} \\
p &: \textit{Anzahl der Propagationsschritte} \qquad p\in\mathbb{N} \\
P(H,\epsilon, g, p) &= P(\underbrace{H_1\cap\ldots\cap H_1}_{p}, \epsilon, g) \\
&= P(H_{1,}\epsilon, g)^p \\
&= \frac{1}{g^p}(g-(2+g-(g+1)\epsilon)P(B,\epsilon, n=g-1, r=0))^p
\end{aligned}
$$

Da B von der Art der Störquellen abhängt, muss für H auch wieder eine getrennte Betrachtung durchgeführt werden.

$$
H_{global} : \textit{clusterübergreifende Propagation bei globalen Störquellen}
$$

$$P(H_{global},\epsilon,g,p) = \frac{1}{g^p}(g-(2+g-(g+1)\epsilon)(1-P(B_{global},\epsilon)))^p$$
$$= \frac{1}{g^p}(g+(g+1)\epsilon^2-(g+2)\epsilon)^p$$

$$H_{local} : \begin{array}{c} \textit{clusterübergreifende Propagation} \\ \textit{bei lokalen Störquellen} \end{array}$$

$$P(H_{local},\epsilon,g,p) = \frac{1}{g^p}(g-(2+g-(g+1)\epsilon)\binom{g-1}{0}(1-\epsilon)^0\epsilon^{g-1})^p$$
$$= \frac{1}{g^p}(g+(g+1)\epsilon^g-(g+2)\epsilon^{g-1})^p$$

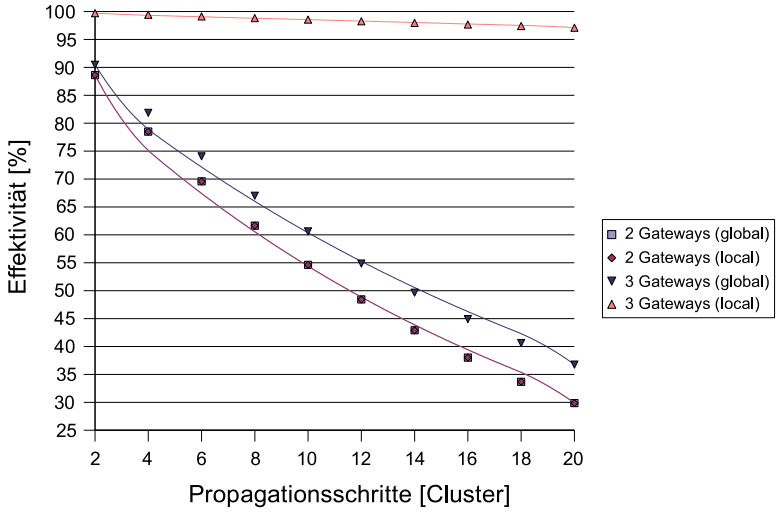


Abbildung 3.7.2d: clusterübergreifende Propagation

Abbildung 3.7.2d zeigt die Ergebnisse bei einer durchschnittlichen Fehlerrate des Netzwerkes von 3%. Bei einer größeren Anzahl Gateways ist unter der Annahme, dass die Störungen nur lokal sind, die Effektivität deutlich höher. Bei globalen Störungen ist der Gewinn mit mehr Gateways nicht so hoch, was auf die Forderung zurückzuführen ist, dass mindestens ein anderer Gateway die Propagation empfangen muss. Die Kernaussage der Ergebnisse ist, dass die Effektivität um so höher ist, je mehr Gateways pro Cluster existieren, dass heißt, je stärker die Cluster untereinander verbunden sind. Existieren mehrere Pfade zwischen Clustern, können Propagationspakete auch über mehrere Pfade die Cluster erreichen. Einzelne Verluste werden so kompensiert.

Der letzte zu untersuchende Kommunikationsprozess ist die **Anfrage** von Cachelines innerhalb eines Clusters. Die Anfrage besteht aus zwei Teilübertragungen. Zuerst versendet ein Knoten ein CellBroadcast mit der Anfrage. Wegen der redundanten Verteilung der Cachelines ist immer eine Teilmenge der Clustermitglieder für die Cacheline verantwortlich. Die Anzahl der zuständigen Knoten ist dabei identisch mit der Anzahl der zuständigen Knoten bei der clusterinternen Propagation. Die Übertragung der Anfrage entspricht somit der Übertragung einer Propagation innerhalb eines Clusters. Die Anfrage ist somit erfolgreich versandt, wenn mindestens ein zuständiger Knoten diese empfangen hat.

$$
\begin{aligned}
I_1 &: \textit{erfolgreiche Übertragung einer Anfrage} \\
P(I_{1,}\epsilon,m) &= P(G,\epsilon,m) \\
&= 1-\left(1-\frac{m\epsilon}{m+1}\right)P(B,\epsilon,n=m,r=0)
\end{aligned}
$$

Die Antwort auf diese Anfrage ist die zweite Teilübertragung. Die Übertragung von einem der zuständigen Knoten zu dem anfragenden Knoten ist eine Punkt-zu-Punkt-Übertragung. Sie entspricht damit dem Ereignis A. Antworten mehrere zuständige Knoten, wäre die gesamte Anfrage erfogreich, wenn nur eine dieser Antworten das Ziel erreicht. Im ersten Schritt wurde aber davon ausgegangen, dass eventuell nur ein zuständiger Knoten die Anfrage erhalten hat. Somit kann bei der Antwort auch nur von einem einzigen Antwortpaket ausgegangen werden.

$$
\begin{aligned}
I_2 &: \textit{erfolgreiche Übertragung der Antwort} \\
P(I_{2,}\epsilon) &= P(A,\epsilon) \\
&= 1-\epsilon
\end{aligned}
$$

Die gesamte Übertragung ergibt sich dann aus der Kombination von I_1 und I_2. Da I_1 von B abhängig ist, muss hier wieder eine Unterscheidung nach den angenommen Störquellen gemacht werden.

$$
\begin{aligned}
I &: \textit{Anfrage} \\
P(I,\epsilon,m) &= P(I_1 \cap I_{2,}\epsilon,m) \\
&= P(I_{1,}\epsilon,m)\cdot P(I_{2,}\epsilon) \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)P(B,\epsilon,n=m,r=0)\right)(1-\epsilon)
\end{aligned}
$$

$$
\begin{aligned}
I_{global} &: \textit{Anfrage bei globalen Störquellen} \\
P(I_{global},\epsilon,m) &= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)(1-P(B_{global},\epsilon))\right)(1-\epsilon) \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)\epsilon\right)(1-\epsilon) \\
&= \frac{1}{m+1}(m+1+(2m+1)\epsilon^2-(m+1)2\epsilon-m\epsilon^3)
\end{aligned}
$$

$$
\begin{aligned}
I_{local} &: \textit{Anfrage bei lokalen Störquellen} \\
P(I_{local},\epsilon,m) &= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)P(B_{local},\epsilon,n=m,r=0)\right)(1-\epsilon) \\
&= \left(1-\left(1-\frac{m\epsilon}{m+1}\right)\epsilon^m\right)(1-\epsilon) \\
&= \frac{1}{m+1}(m+1+(2m+1)\epsilon^{m+1}-(m+1)(\epsilon+\epsilon^m)-m\epsilon^{m+2})
\end{aligned}
$$

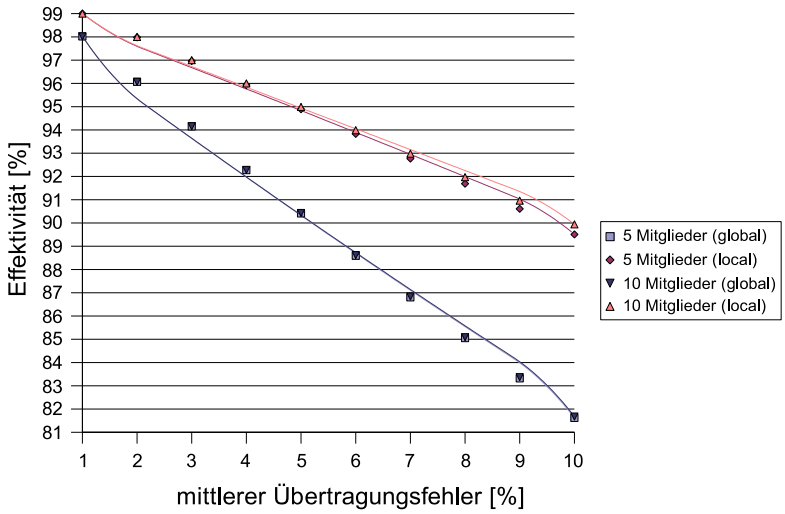


Abbildung 3.7.2e: Anfrage innerhalb eines Clusters

In Abbildung 3.7.2e sind die Ergebnisse für die Effektivität der Kommunikation bei Anfragen innerhalb eines Clusters dargestellt. Der Erfolg der Übertragung ist hier direkt von der durchschnittlichen Fehlerrate des Netzwerkes abhängig. Unter der Annahme globaler Störquellen ist es sogar das Doppelte der Fehlerrate, da beide Teilübertragungen dann direkt von der Fehlerrate abhängen. Unter der Annahme von lokalen Störquellen ist nur die Antwort direkt abhängig. Bei der Anfrage kann die Redundanz des Systems ausgenutzt werden. Eine Abhängigkeit von der Anzahl der Mitgliedsknoten ist nur bei lokalen Störquellen minimal auszumachen. Eine Verbesserung ist nur durch Übertragungswiederholungen erreichbar.

Als Ergebnis der Analyse lässt sich sagen, dass das entwickelte Cachesystem die ungünstigen Voraussetzungen optimal ausnutzt. Obwohl Pakete verloren gehen können und keine Komplementierung des Verlustes möglich ist, werden meist Ergebnisse von über 90% Effektivität erreicht. Bei der Synchronisierung wurde gezeigt, dass mit Übertragungswiederholungen an schwachen Stellen der Kommunikation erhebliche Verbesserungen zu erreichen sind, dabei die Bandbreite aber trotzdem nur minimal zusätzlich belastet wird. Eine allgemeine Verbesserung lässt sich durch adaptive Fehlertoleranz herstellen. Anhand der Ausprägung der Qualität der Datenübertragung, werden Pakete dann mit angepasster Häufigkeit versendet. Dies muss in der Netzwerk/Routing-Schicht geschehen. Die benötigten Informationen können dort von Monitoringkomponenten aus Daten von Übertragungsmethoden gezogen werden, die Rückmeldungen über den Erfolg von Übertragungen beinhalten. Zu diesen Methoden gehört auch der Reliable CellBroadcast. Dieser wird im gesamten Cachesystem zwar nicht benötigt, allerdings ist die Nutzung in anderen Komponenten der PSMW wahrscheinlich.

4 Implementierung

Dieses Kapitel beschäftigt sich mit der Implementierung des Cachesystems. Als Erstes wird die Speicherstruktur des Caches auf den einzelnen Knoten implementiert. Danach wird die Verteilung und Replikation innerhalb eines Clusters und anschließend die Propagation über Clustergrenzen hinweg implementiert. In Abbildung 4 ist eine grober Überblick der entstehenden Struktur zu sehen.

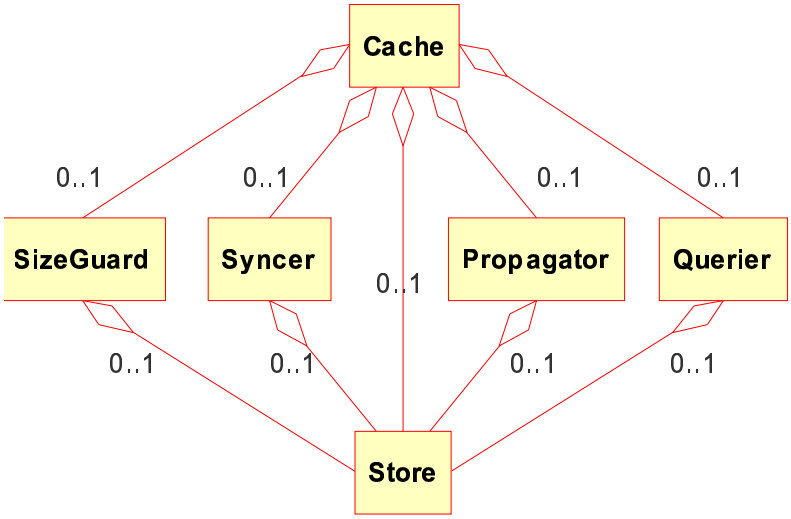


Abbildung 4: grobe Systemstruktur

Der Cache wird aus fünf Modulen zusammengesetzt, die in den folgenden Kapiteln näher beschrieben werden. Die Basis des gesamten Caches stellt der Store dar. Er ist die Speicherstruktur einer jeden Instanz des Caches. Die anderen vier Module SizeGuard, Syncer, Propagator und Querier nutzen ein und dieselbe Instanz des Stores. So ist es möglich, die einzelnen Module unabhängig voneinander zu implementieren. Das Cachesystem ist dadurch später leicht erweiterbar. Der SizeGuard ist für die Überwachung und Pflege des benötigten Speichers des Stores verantwortlich. Der Syncer übernimmt die in Kapitel 3.4 erklärte Synchronisation der Daten. Der Propagator ist für die clusterinterne und clusterübergreifende Propagation aus Kapitel 3.5 zuständig. Und der Querier befragt die Mitglieder des Clusters nach bestimmten Cachelines und beantwortet die Anfragen anderer Mitglieder. Der Cache ist dann noch für die Verteilungskonfiguration und als Schnittstelle für Applikationen zuständig.

4.1 Speicherstruktur/Store

Die kleinste Einheit des Systems wird durch eine **Cacheline** gebildet. Sie stellt einen Eintrag im Cache dar. Der Inhalt der Daten ist für den Cache nicht relevant. Sie werden ohne jegliches Wissen über Verwendung, Zweck und Wichtigkeit verarbeitet. Zur Identifikation erhält jede Cacheline neben den Daten, die als Byte Array gespeichert werden, einen eindeutigen Schlüssel. Des weiteren werden drei Zeitstempel für jede Cacheline benötigt. **Mtime** ist die „modification time“. Er stellt

den Zeitpunkt der Erstellung oder der letzen Aktualisierung der Daten dar. **Atime** ist die „access time“. Er ist der Zeitpunkt, zu dem das letzte Mal auf diese Cacheline zugegriffen wurde. Über dieses Attribut wird die Ersetzung von Cachelines nach dem Least-Recently-Used-Verfahren entschieden (siehe Kapitel 3.3). **Maxage** stellt das maximale Alter einer Cacheline dar. Das Alter wird mit Hilfe von mtime ermittelt. Wird maxage überschritten, ist die Cacheline nicht mehr verwendbar und wird unverzüglich gelöscht. Die Operation **isToOld()** übernimmt hierbei die Altersüberprüfung. Abbildung 4.1 zeigt ein entsprechendes Klassendiagramm.

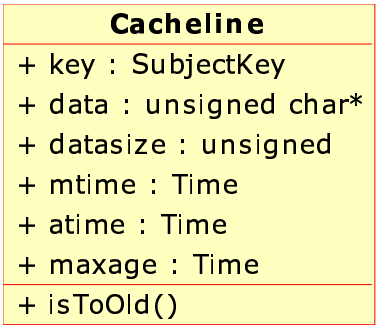


Abbildung 4.1: Klassendiagramm Cacheline

Auf die Daten des Caches, eine Menge von Cachelines, wird auf zwei Wegen zugegriffen. Die Suche im lokalen Speicher findet anhand des Schlüssels statt. Um optimale Suchgeschwindigkeiten zu erreichen, kommt ein B-Baum mit Hilfe des STL Templates **map** zum Einsatz. Der Aufwand für Suchen, Einfügen und Löschen ist dadurch logarithmisch. Die Implementierung der LRU Ersetzungsstrategie benötigt einen schnellen Zugriff auf die Cachelines mit den ältesten Zugriffszeiten. Da der Baum anhand des Schlüssels aufgebaut ist, müsste hier jedesmal der gesamte Baum durchsucht werden. Um den Aufwand für die Ersetzungsstrategie zu minimieren wird zusätzlich eine Liste benötigt, in der die Cachelines abfallend nach der „access time“ sortiert sind. Das Löschen zu ersetzender Cachelines kann dann mit konstantem Aufwand durchgeführt werden. Einfügungen in die Liste benötigen auf Grund der Sortierung linearen Aufwand. Um beide Strukturen unterstützen zu können, werden jeweils nur Zeiger auf die Cachelines gespeichert, so dass die Speicheranforderung für eine Cacheline unabhängig ist vom Baum bzw. der Liste. In Abbildung 4.1a ist die lokale Speicherung mit beiden Zugriffspfaden dargestellt.

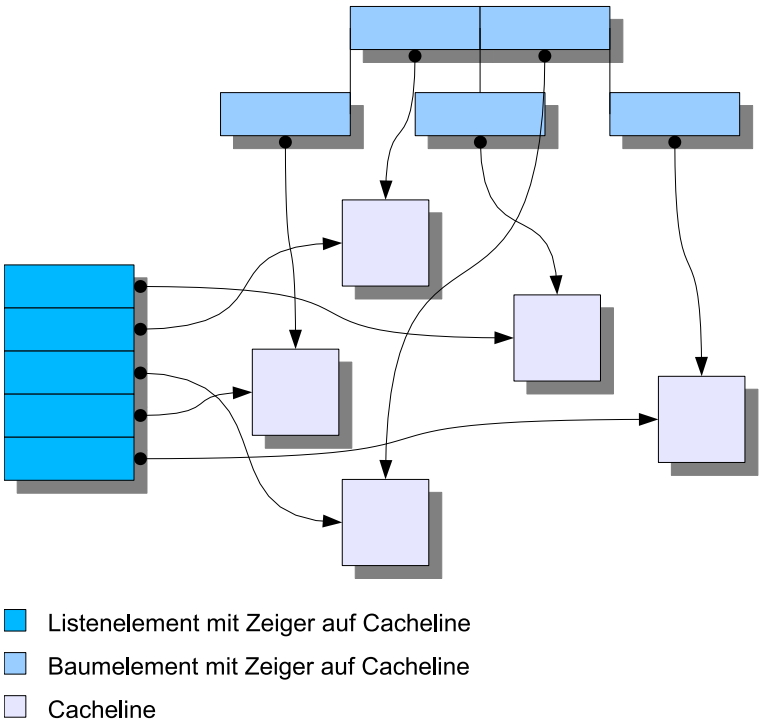


Abbildung 4.1a: lokale Speicherstruktur

Da nun zwei Zugriffspfade existieren, müssen auch beide bei jeder Operation gepflegt werden. Die Folge ist ein addierter Aufwand bei den Operationen des Store. Das Einfügen ist damit linear, weil hier der logarithmische Aufwand beim Baum und der lineare Aufwand der sortierten Liste in Anspruch genommen wird. Beim Löschen wird das letzte Element der Liste gelöscht, der Aufwand ist somit konstant. Um die Cacheline auch aus dem Baum zu löschen, wird ein logarithmischer Aufwand benötigt. Zusammen ergibt sich somit ein logarithmischer Aufwand für das Löschen. Ein dritter Anwendungsfall ergibt sich aus dem Least-Recently-Used-Verfahren. Hat eine Cacheline einen Hit, wird der atime Zeitstempel neu gesetzt. Dadurch ist die Liste der Cachelines aber nicht mehr sortiert. Um die Liste wieder in einen sortierten Zustand zu überführen, muss die falsch sortierte Cacheline durch einfache Listensuche an den richtigen Platz verschoben werden. Für diese Umsortierung ergibt sich somit ein linearer Aufwand. Allerdings werden Cachelines im Allgemeinen über den Baum gesucht, somit muss eine Cacheline erst in der Liste nochmal gesucht werden, bevor eine Neusortierung stattfinden kann. Da dies den Vorteil der logarithmischen Suche des Baumes außer Kraft setzt ist es nicht akzeptabel. In Abbildung 4.1b ist ein erweitertes Klassendiagramm der Cacheline zu sehen. Jede Cacheline hat zusätzlich einen Iterator, der auf das zugehörige Listenelement zeigt. Mit Hilfe dieser doppelten Verkettung zwischen Cacheline und Listenelement wird das Problem beseitigt, da das wieder richtig einzusortierende Element nicht erst in der Liste gesucht werden muss.

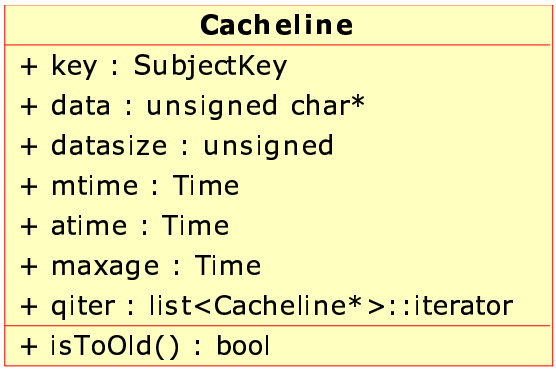


Abbildung 4.1b: erweitertes Klassendiagramm Cacheline

In Kapitel 3.1 ist die Verteilung der Cachelines im Cluster beschrieben. Basis der Verteilung ist das **Consistent-Hashing**. Die kleinste Verteilungseinheit ist hier die Menge Cachelines, die denselben Wert einer fest definierten Hashfunktion haben. Die verwendete Hashfunktion ist:

$$h(k) \quad = \quad k \bmod 256$$

Pro Cache existieren somit 256 solcher Einheiten, die im Folgenden **Bucket** genannt werden. Zu beachten ist, dass in Kapitel 3 bei der Vorstellung des Verfahrens ein Bucket ein Container für die lokal gespeicherten Verteilungseinheiten darstellt. Solch ein Container ist für die Implementierung aber nicht notwendig. Die Bezeichnung Bucket für die Verteilungseinheiten selbst, erweist sich dafür angebracht. Die Speicherstruktur der Cachelines muss in Bezug auf das Bucketkonzept aus zwei Richtungen betrachtet werden. Einleuchtend ist, dass jedes Bucket in einer eigenen Struktur abgelegt ist. Dagegen spricht allerdings die Least-Recently-Used-Ersetzungsstrategie. Wird je Bucket eine LRU-Queue genutzt, kann es vorkommen, dass Cachelines in stark genutzten Buckets paradoxerweise viel zeitiger gelöscht werden. Oder um die am längsten nicht genutzte Cacheline zu finden, müssen erst die letzten Cachelines aller Bucketqueues verglichen werden. Naheliegend ist somit, die LRU-Queue für alle Buckets zu halten. Cachelines können dann unabhängig vom Bucket ersetzt werden. Abbildung 4.1c skizziert die so entstehende Speicherstruktur.

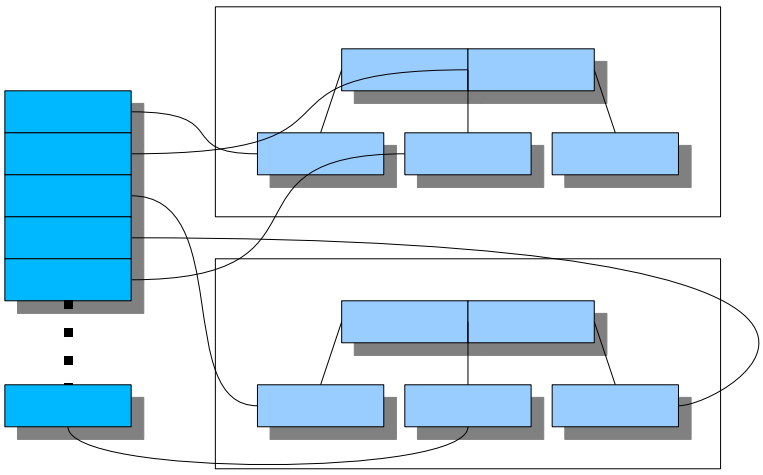


Abbildung 4.1c: entgültige Speicherstruktur (vereinfachte Darstellung)

Die Klasse **Store** repräsentiert die Speicherstruktur des Caches. Jeder Knoten des Caches hat genau eine Instanz dieser Klasse. Abbildung 4.1d zeigt das

Klassendiagramm von Store.

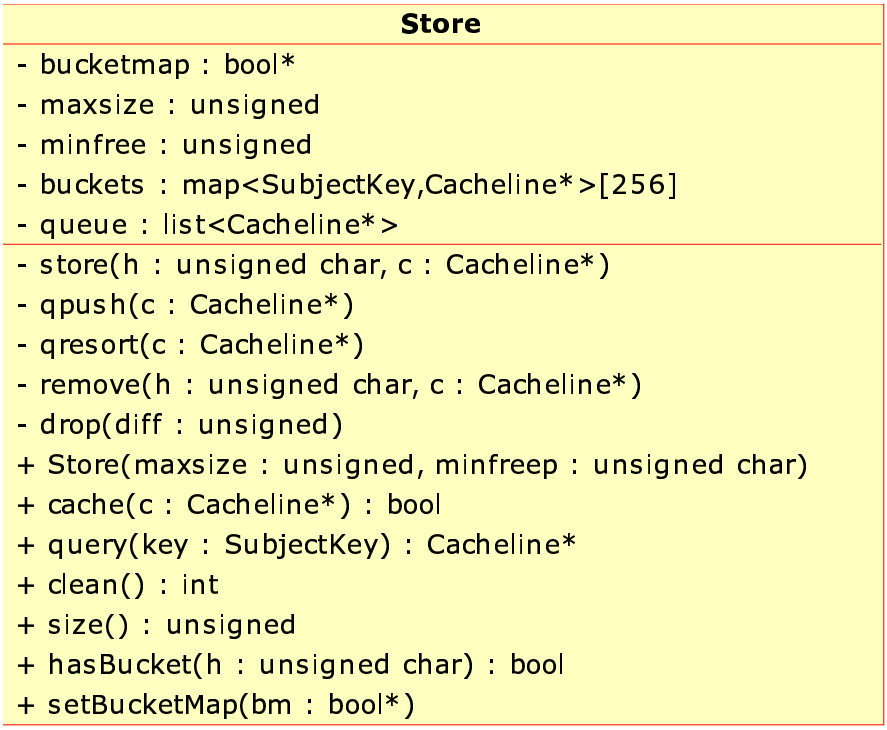


Abbildung 4.1d: Klassendiagramm Store

Ein **Store**-Objekt hat neben dem Array **buckets** mit den 256 Baumstrukturen der Buckets und der LRU-Queue (**queue**), die die Cachelines aller Buckets enthält, noch eine **bucketmap** und mit **maxsize** und **minfree** Angaben zur Steuerung der Größe des Store-Objektes. Bucketmap ist ein Array aus 256 Wahrheitswerten, in denen hinterlegt ist, ob der lokale Cache die entsprechenden Buckets besitzt. Alle Buckets, deren Wert dort **false** ist, sind auf anderen Knoten im Cluster gespeichert. Durch maxsize ist die maximale Anzahl an gespeicherten Cachelines definiert. Ist diese Grenze erreicht, muss beim Einfügen einer neuen Cacheline erst eine andere Cacheline mit Hilfe der LRU-Queue gelöscht werden. Um den Aufwand beim Einfügen zu reduzieren, gibt minfree die Anzahl von nicht belegten Cachelines an. Die Operation **clean()** überprüft die Anzahl gespeicherter Cachelines. Ist diese größer als $maxsize - minfree$ werden so viele Cachelines wie nötig gelöscht, um die freizuhaltenden Platz wieder herzustellen. clean() wird später vom SizeGuard aufgerufen. Dadurch wird erreicht, dass minfree Cachelines eingefügt werden können, ohne dass erst alte nach dem LRU Verfahren gelöscht werden müssen. In Situationen mit einer großen Anzahl neuer Cachelines wird der Aufwand der Einfügungen dadurch deutlich reduziert. Es entstehen geringere Latenzzeiten bei der Bereitstellung der Daten. Solche Burstsituationen mit großen Mengen an zu speichernden Cachelines entstehen bei Bewegungen von Knoten zwischen Clustern recht häufig, da bei der Integration in einen neuen Cluster die Daten synchronisiert werden. Einzelheiten dazu werden in Kapitel 4.3 näher erläutert. Im Aktivitätsdiagramm in Abbildung 4.1e ist der Ablauf einer Einfügung dargestellt.

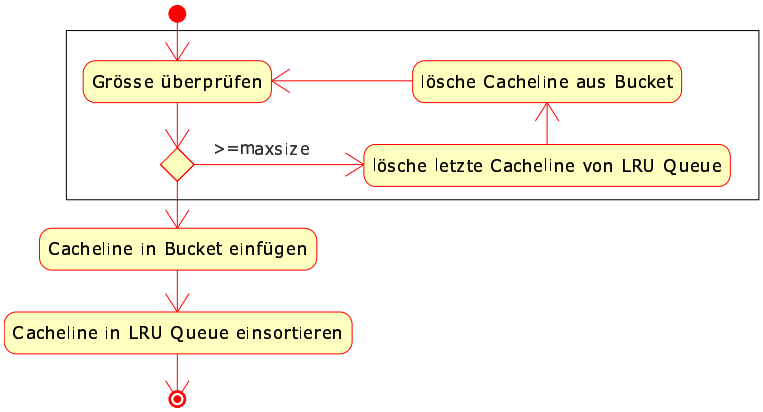


Abbildung 4.1e: Aktivitätsdiagramm Einfügen im Store

Mit der Freihaltung eines Teils des verfügbaren Speicherplatzes muss der eingerahmte Teil nur selten vollständig ausgeführt werden. In der zusätzlichen Schleife ensteht ein konstanter Aufwand für das Löschen aus der Queue und ein logarithmischer Aufwand für das Löschen aus dem Bucket. Bei einem vollen Store wird dieser Aufwand zusätzlich zu dem Aufwand für die beiden Einfügeoperation benötigt. Beim Einfügen einer größeren Menge von Cachelines summiert sich der zusätzliche Aufwand dann pro Einfügung.

Eine Besonderheit des Caches gegenüber einem Datenspeicher ist die Handhabung der Alterung und Ersetzung. Da Letzteres über die LRU-Queue realisiert wird, die absteigend nach dem Zeitstempel **atime** sortiert ist, werden Hits, also erfolgreiche Anfragen an den Cache, gesondert behandelt, da sie eine Neusortierung der Queue nach sich führen. Abbildung 4.1f zeigt den Ablauf einer Anfrage.

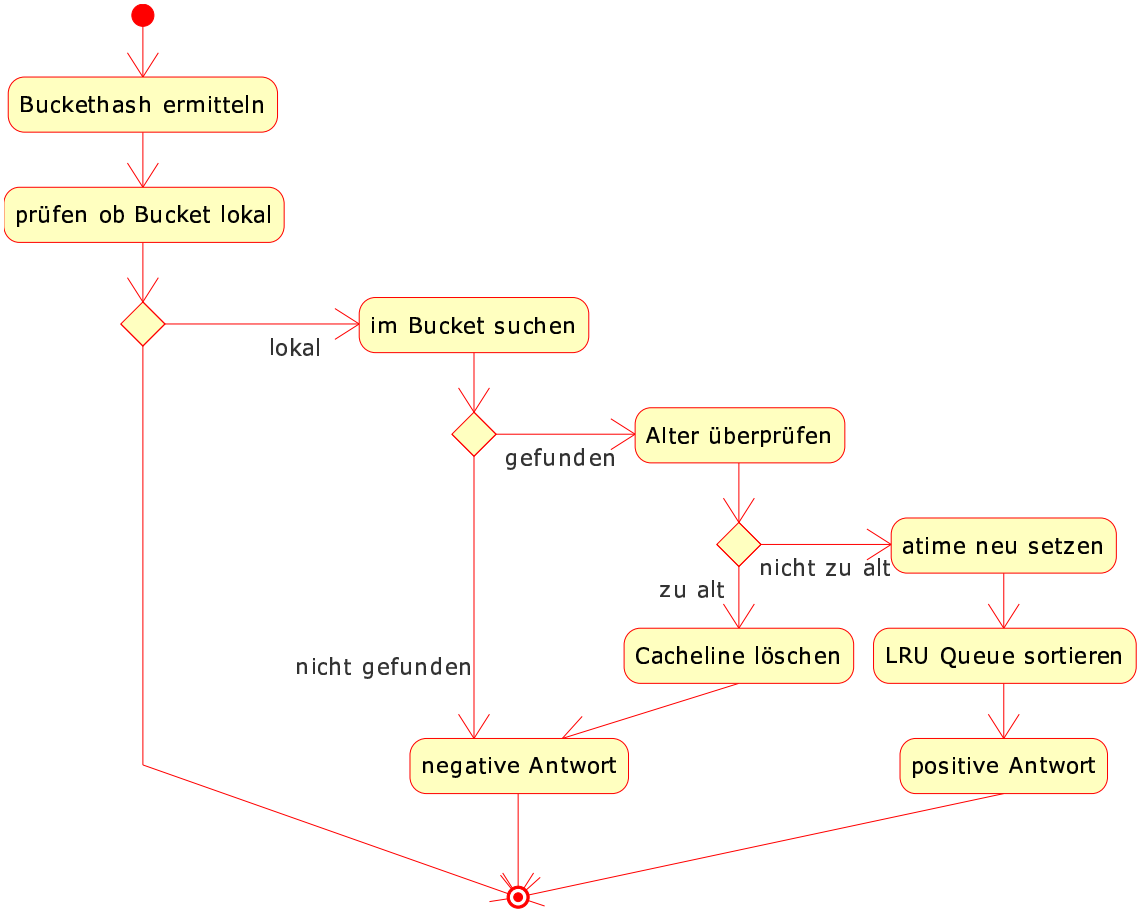


Abbildung 4.1f: Ablauf Anfrage

Die Basis der Verteilung, die im nächsten Kapitel besprochen wird, ist in Abbildung 4.1f beispielhaft für alle Operationen des Caches dargestellt. Über die **bucketmap** wird ermittelt, ob das Zielbucket lokal gepflegt wird. Ebenfalls beispielhaft für alle Operationen ist die Handhabung des Alters. Cachelines mit überschrittenem **maxage** werden als nicht existent behandelt und gegebenenfalls sofort gelöscht. In Abbildung 4.1g ist dementsprechend der Ablauf der zweiten Hauptoperation auf dem Store, das Cachen, dargestellt.

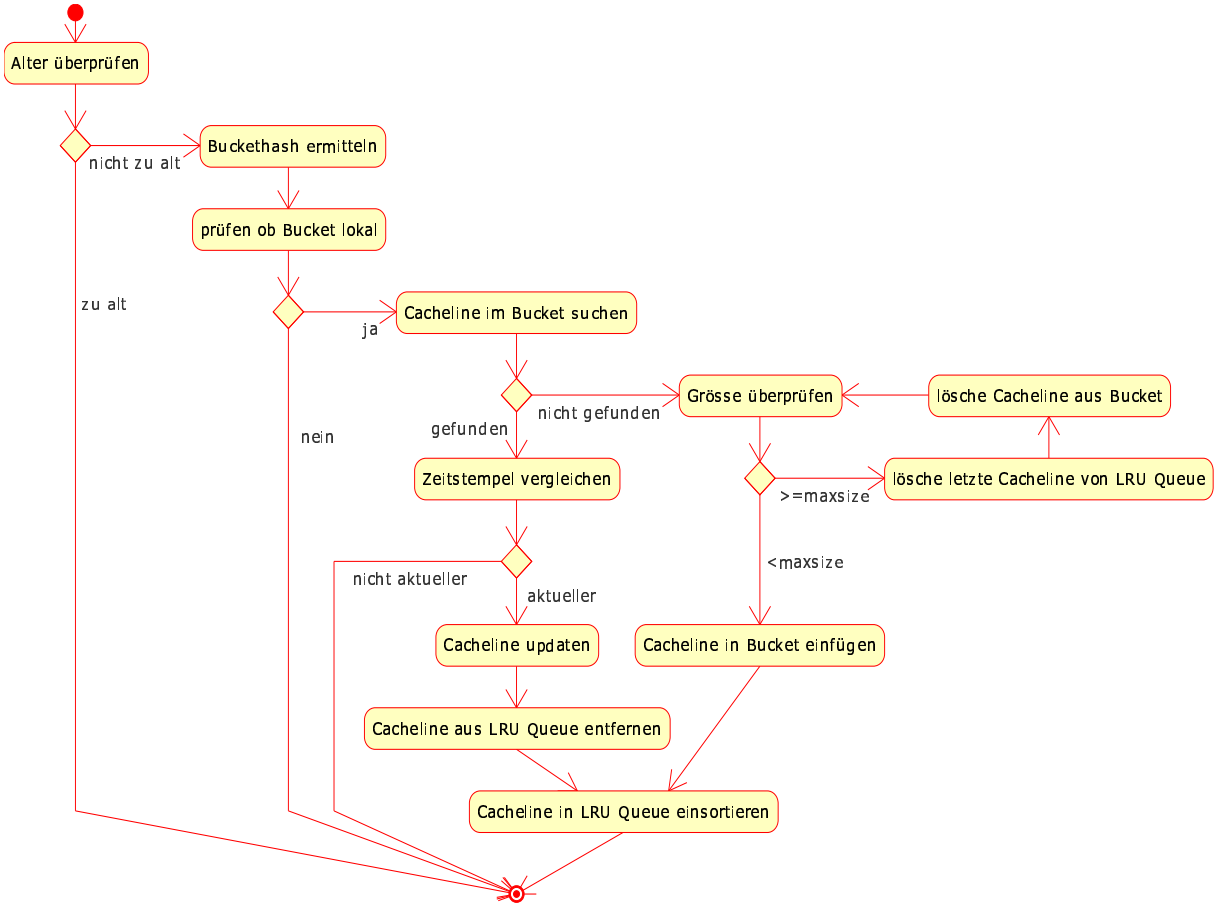


Abbildung 4.1g: Ablauf Cachen

4.2 Kommunikation

Für die Kommunikation aller Module des Cachesystems sind zwei Klassen zuständig. Der **Sender** ist für das Versenden der CellBroadcastpakete zuständig. Analog dazu ist der **Receiver** für den Empfang der Pakete und den Aufruf der entsprechenden Handlermethoden der Module verantwortlich. Die gesamte Netzwerkkommunikation basiert auf den Best-Effort-Methoden der PSMW. Abbildung 4.2 zeigt ein Klassendiagramm beider Klassen.

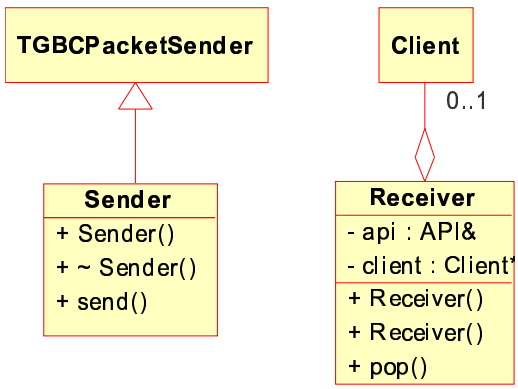


Abbildung 4.2: Klassendiagramm Sender und Receiver

Die Superklasse von Sender ist TGBCPacketSender. Es ist ein Klasse aus dem **utils** Paket der PSMW. Die Funktion des TGBCPacketSender ist das Versenden von Broadcastpaketen. Für die Nutzung im Cachesystem wurde diese nur um eine Methode erweitert, die es ermöglicht, Paketdaten in Form eines Byte Arrays zum Senden zu übergeben. Der Receiver benutzt ein Objekt der Klasse Client. Diese ist aus dem Routing API Paket (**rapi**) der PSMW. Es handelt sich dabei um einen Empfänger, der mit einer Handlermethode initialisiert wird, die die empfangenen Pakete verarbeitet.

4.3 Verteilung

Nachdem in Kapitel 4.1 die Speicherstruktur des Caches auf einzelnen Knoten festgelegt wurde, ist der Fokus jetzt auf die Verteilung der Cachelines innerhalb eines Caches gerichtet. Wie in Kapitel 3 beschrieben, wird ein Cache durch genau einen Cluster im Netz gebildet. Die Koordination wird somit naheliegend vom Clusterhead erledigt. Er hat die Aufgabe, allen vorhandenen Mitgliedern des Clusters eine Konfiguration zuzuweisen. Diese Konfiguration besteht aus einer Liste aller Mitglieder und dem primären Bucket, dem sie zugewiesen werden. Der Clusterhead übernimmt an dieser Stelle die Konfiguration des **Consistent Hashing**. Abbildung 4.3 zeigt die Klasse CacheConfig, die die Verteilungskonfiguration eines Clusters darstellt. Der Clusterhead versendet diese per CellBroadcast wie in Kapitel 3.2.4 beschrieben.

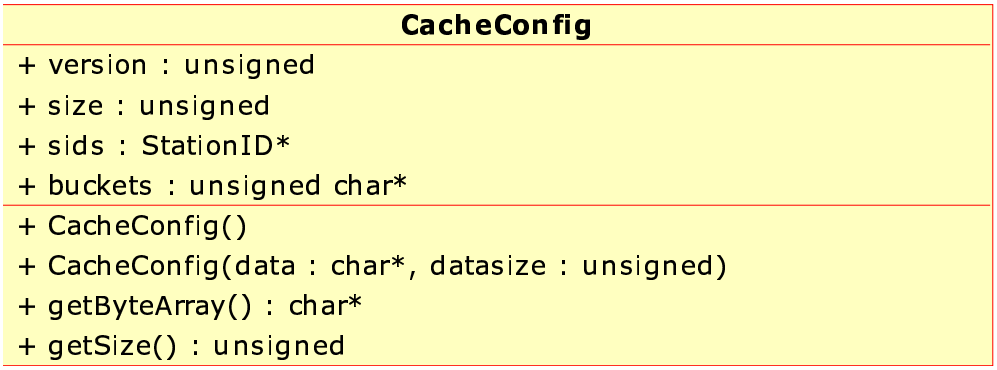


Abbildung 4.3: Klassendiagramm CacheConfig

Anhand des primären Buckets können die Knoten eines Clusters den Zuständigkeitsbereich über der Menge der Buckets ermitteln.

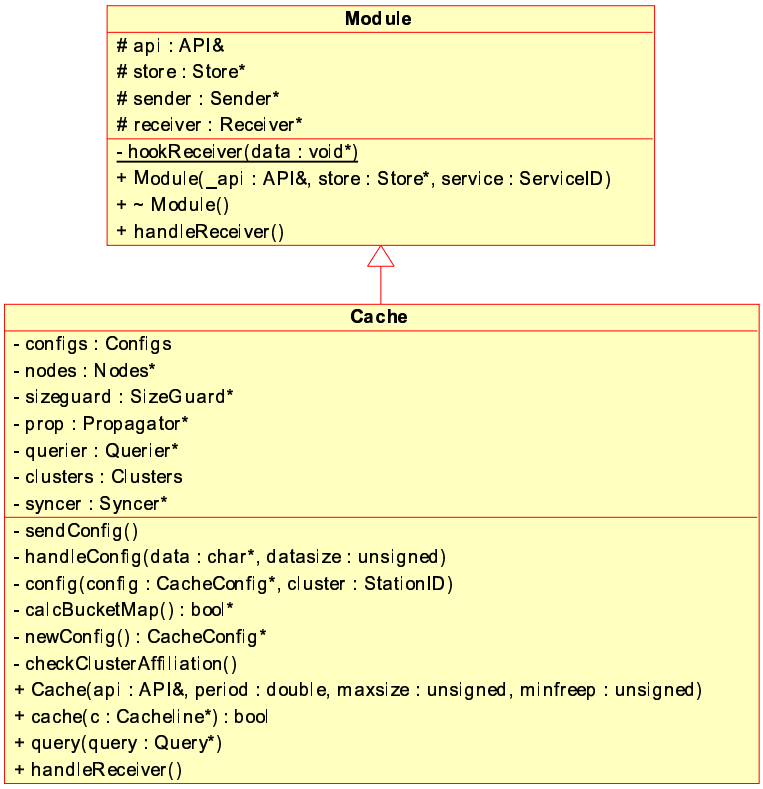


Abbildung 4.3a: Klassendiagramm Cache

Abbildung 4.3a zeigt das Klassendiagramm der Klasse Cache. Da Funktionen und Objekte, wie der Zugriff auf die Schnittstelle der PSMW, der Store, der Sender und der Receiver, von allen Modulen des Cachesystems gleichermaßen benötigt werden, wurde diese in die Klasse Module eingebaut, die die Superklasse aller Module ist. Wichtig für die Verteilung sind in der Klasse Cache die Methoden sendConfig(), handleConfig(), config(), newConfig() und checkClusterAffialiation(). **checkClusterAffialiation()** wird auf dem Clusterhead durch das Ereignis einer neuen Clusterkonstellation, das von der PSMW erzeugt wird, aufgerufen. Da sich der Cache die Konstellation des Clusters merkt, wird nun ein Vergleich der neuen und der alten durchgeführt. Fällt der Vergleich negativ aus, wird mit Hilfe von **newConfig ()** eine neuen Konfiguration des Clusters gebildet. Das Senden der Konfiguration geschieht mit **sendConfig()**. Durch das Ereignis einer empfangenen Konfiguration wird auf den Clients dann **handleConfig()** aufgerufen. Diese konfiguriert die Cacheinstanz dann mit der Methode **config()**. Auf dem Clusterhead muss die Konfiguration nicht erst empfangen werden, da dieser sie selbst erstellt und versendet. config() wird hier schon von checkClusterAffialiation() aufgerufen, nachdem die Konfiguration ermittelt wurde.

Für die Interaktion mit Applikationen sind die beiden Methoden cache() und query() zuständig. Über **cache()** können Applikationen Cachelines in das System eintragen. Ist die lokale Instanz für das entsprechende Bucket verantwortlich, wird die Cacheline im Store gecached. Anschließend wird die Cacheline an den Propagator weitergereicht, der sie dann propagiert. Der Propagator wird später noch beschrieben. Anfragen an das Cachesystem werden von Applikationen mit Hilfe der Methode **query()** gestellt. Dabei wird nicht nur der Schlüssel der gesuchten

Cacheline sondern auch eine Callbackfunktion der Applikation übergeben. Da das System vollständig asynchron arbeitet sobald eine Anfrage an andere Mitglieder des Clusters gestellt werden muss, weil die lokale Instanz nicht für das entsprechende Bucket zuständig ist, kann auch nur asynchron auf eine Anfrage reagiert werden. Muss die Anfrage an andere Clustermitglieder weitergereicht werden, übernimmt dies der Querier, der wie der Propagator, auch erst später beschrieben wird.

4.4 Syncer

Ändert sich die Zuständigkeit eines Knotens für ein Bucket, können zwei Situationen entstehen. Bekommt der Knoten ein zusätzliches Bucket hat er im Allgemeinen dort noch keine Daten gespeichert. Andere Knoten, die bereits für dieses Bucket zuständig sind, haben dagegen bereits Daten. Eine Synchronisation muss stattfinden. Für die Synchronisation ist ein eigenes Objekt, der **Syncer**, zuständig. Die Synchronisation ist in Kapitel 3.4 detailliert beschrieben. In Abbildung 4.4 ist der ungestörte Ablauf der Synchronisation in einem Sequenzdiagramm dargestellt.

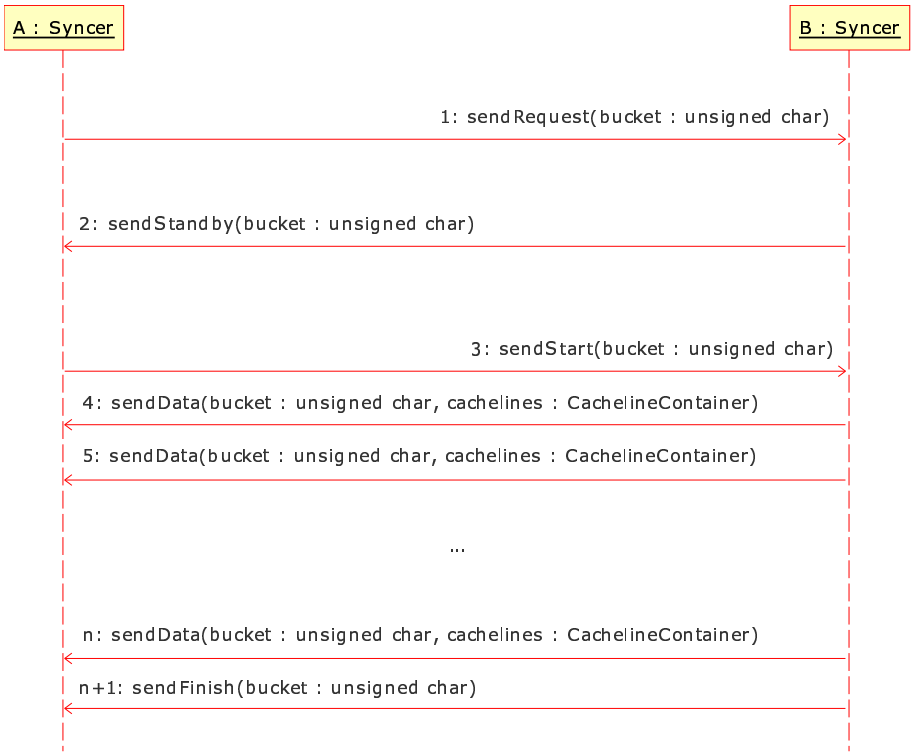


Abbildung 4.4: Sequenzdiagramm Synchronisation

Bei Verlust der Zuständigkeit für ein Bucket wird das Bucket im Store nicht geleert. Es wird so vermieden, dass bei der Leerung eines Buckets ein hoher Aufwand für das Löschen der entsprechen Einträge aus der Queue entsteht. Nachteilig daran ist die Verschwendung von Speicherplatz. Durch die bucketübergreifende LRU-Queue altern die Cachelines aber mit der Zeit heraus. Positiv ist allerdings, dass nicht gemanagte Buckets, die noch Cachelines haben, bei erneuter Zuständigkeit bereits lokal Ergebnisse liefern können, bevor synchronisiert wurde. Dieser Vorteil macht sich besonders bei Netzen mit hoher Mobilität bemerkbar.

Um die Synchronisationen eindeutig unterscheiden zu können, dient die Kombination aus Bucket und ID des Knotens, der die Synchronisationsdaten verschickt. Somit kann es nicht zu Situationen kommen, in denen mehr als ein Knoten die Bereitschaft zum Senden mit **Standby** signalisiert, und durch ein **Start** Signal des anfragenden Knotens parallel dieselben Daten versenden. Abbildung 4.3a zeigt die Klasse **SyncPacket**, die die Basis der Synchronisationskommunikation darstellt.

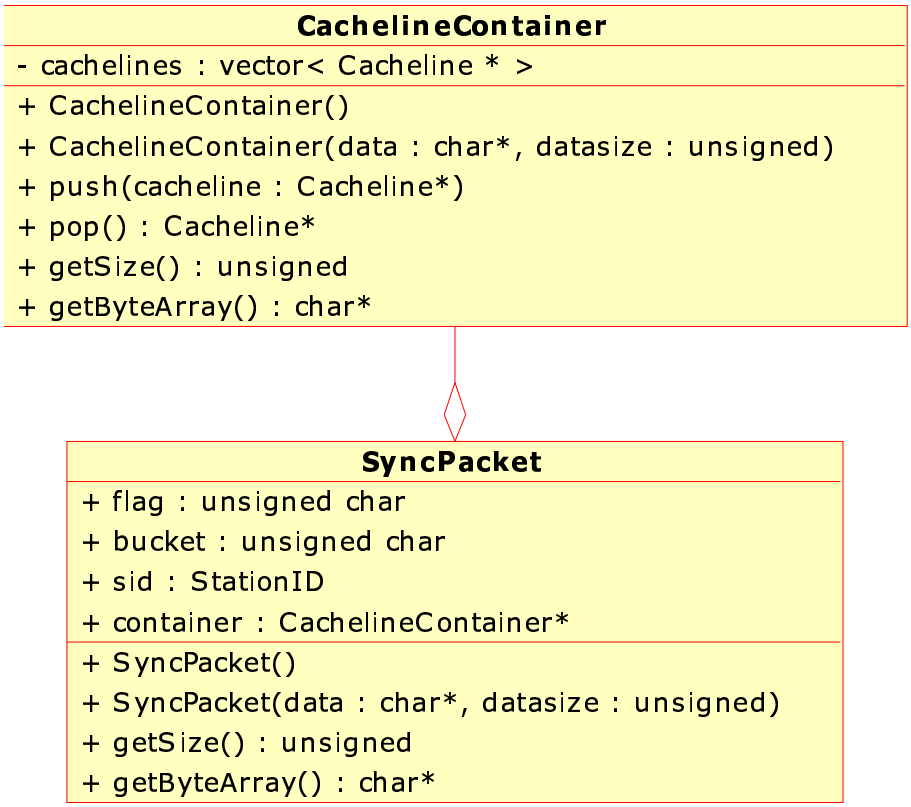


Abbildung 4.4a: Klassendiagramm SyncPacket

In den zwei folgenden Abbildungen ist der Synchronisationsprozess aus Sicht des Synchronisationssenders (Abbildung 4.4b) und des Synchronisationsempfängers (Abbildung 4.4c) dargestellt.

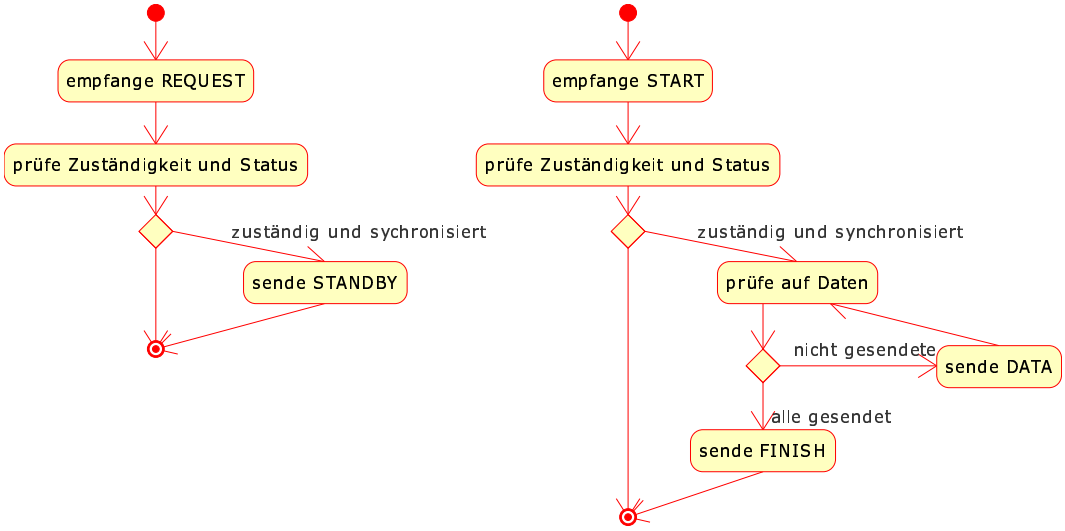


Abbildung 4.4b: Aktivitätsdiagramm Synchronisationssender

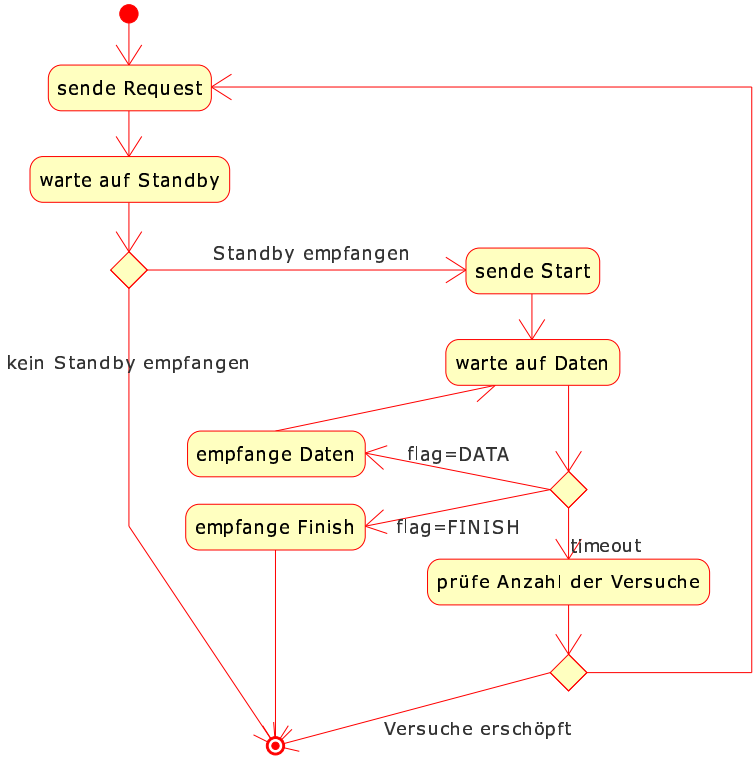


Abbildung 4.4c: Aktivitätsdiagramm Synchronisationsempfänger

Die ereignisbasierte Verarbeitung zeigt sich beim Syncer deutlich in der Menge der Methoden. Für jedes Kommunikationsereignis existiert eine Handlermethode, die an der Namesgebung **handle*()** zu erkennen sind. Mit den **send*()** Methoden werden die entsprechenden Pakete versendet. Abbildung 4.4d zeigt das Klassendiagramm der Klasse Syncer.

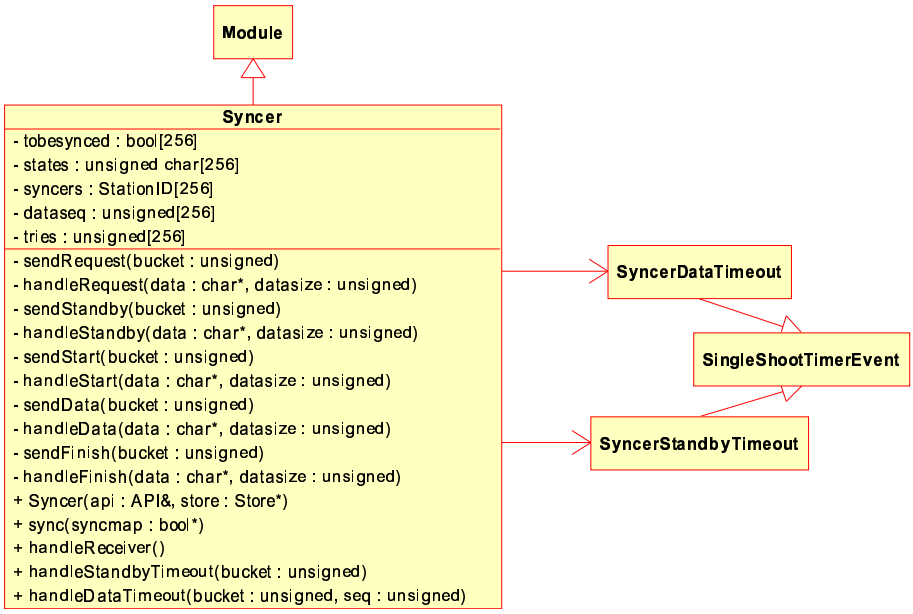


Abbildung 4.4d: Klassendiagramm Syncer

Bei der Kommunikation werden Time-outs verwendet. Realisiert sind diese als Subklasse der Klasse SingleShootTimerEvent. Es ist eine Ereignisklasse von GEA. Das Resultat sind die beiden Klassen **SyncerStandbyTimeout** und **SyncerDataTimeout**. Bei der Initialisierung von Instanzen dieser Klassen registrieren sie sich bei GEA als Ereignis, das zu einer bestimmten Zeit aktiviert wird. Dabei werden dann die Methoden **handleStandbyTimeout()** bzw. **handleDataTimeout()** aufgerufen, die dann die nötigen Aktionen wie Neubeginn der Synchronisation erledigen.

4.5 Propagator

Die Synchronisation aus dem vorherigen Kapitel sorgt für gleiche Datenbestände der Buckets auf den unterschiedlichen Knoten eines Clusters. Dies geschieht immer bei Änderungen der Konstellation eines Clusters. Für die Verbreitung der Daten im Normalbetrieb ist der Propagator zuständig. Neue und geänderte Cachelines werden ihm übergeben. Unabhängig von anderen Modulen des Cachesystems verbreitet, oder passender formuliert, propagiert er diese aktuellen Daten. Der **Propagator** arbeitet wie auch zuvor schon der Synchronisator und der Cache selbst zur Konfiguration nur mit CellBroadcast. Da diese vom Clusterhead zu allen Mitgliedern des Clusters übertragen werden, verbreiten sich alle Änderungen immer im gesamten Cluster.

Der Propagator hat zwei Möglichkeiten, Cachelines zu propagieren. In der ersten Variante werden Cachelines direkt propagiert. Intuitiver Vorteil dieses Verfahrens ist die damit verbundene schnelle Verfügbarkeit von Daten im gesamten System. Daraus resultiert aber auch gleich der große Nachteil. Es müssen sehr häufig Datenpakete über das Netzwerk transportiert werden. Da die Anforderung an die Konsistenz der Datenbestände hier relativ gering angesiedelt ist, müssen geänderte Cachelines nicht sofort propagiert werden. In der zweiten Propagationsvariante hat

der Propagator eine Queue, in der die Cachelines gespeichert werden. In periodischen Abständen wird der Inhalt der Queue propagiert. Die ständige Belastung des Netzes mit Datenpaketen ist somit eingedämmt. Mehrere Cachelines können nun sogar gebündelt übertragen werden. Die Bandbreite des Netzes wird somit besser genutzt. Für die Bündelung wird die Klasse **CachelineContainer** eingesetzt. Abbildung 4.5 zeigt das entsprechende Klassendiagramm.

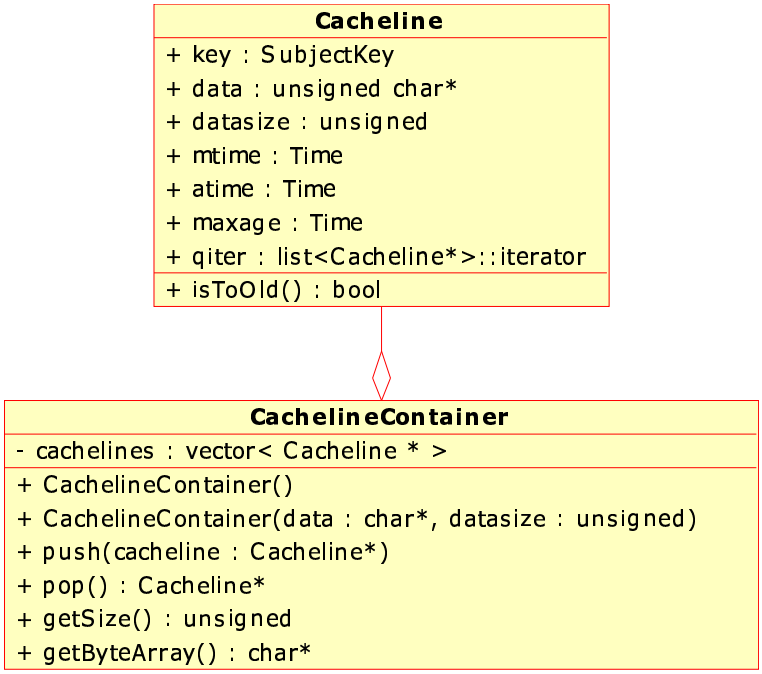


Abbildung 4.5: Klassendiagramm CachelineContainer

Mit den Methoden **push()** und **pop()** werden dem Container Cachelines zugefügt bzw. entnommen. **getByteArray()** gibt eine serialisierte Version des Containers als Byte Array aus. Die Größe dieses Arrays lässt sich mit **getSize()** ermitteln. Auf diese Weise lässt sich die Größe eines Containers an den verfügbaren Platz in einem CellBroadcastpaket anpassen.

Die Bündelung von direkten Propagationen ist nicht möglich, da eben genau eine Cacheline zum Zeitpunkt der Propagation verbreitet werden soll. Um aber nicht ein Paket mit nur einer Cacheline zu versenden, wird die direkt zu propagierende Cacheline mit Cachelines aus der Queue gebündelt. So wird das Paket maximal ausgelastet. Abbildung 4.5a zeigt den Ablauf einer direkten Propagation.

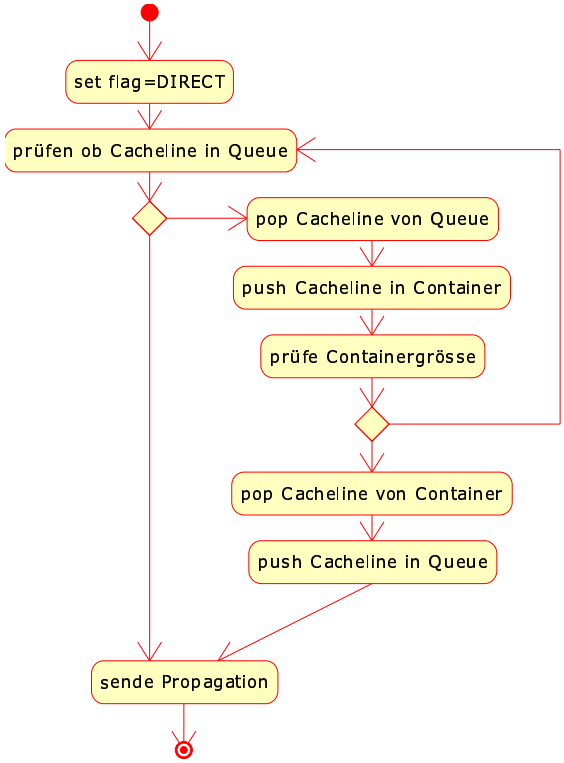


Abbildung 4.5a: direkte Propagation

Viele Änderungen betreffen aber Cachelines, die bereits existieren. Die wohl häufigste Änderung ist in diesem Zusammenhang die Aktualisierung der **atime** einer Cacheline. Dies geschieht bei jedem erfolgreichen Zugriff auf eine Cacheline (Hit). Diese Cachelineänderungen werden verzögert propagiert und können somit gebündelt, bzw. zum Bündeln mit direkten Propagationen genutzt werden, um die Bandbreite des Netzes zu schonen. Zum Einsatz kommt dabei die bereits erwähnte Queue. Neben dem Bündeln mit direkten Propagationen werden sie in periodischen Abständen komplett versendet. Damit die Queue sich nicht unendlich füllen kann, werden dann alle Cachelines propagiert. Es ist somit wahrscheinlich, dass mehr als ein Paket dabei versendet wird. Der Ablauf der **queued Propagation** ist in Abbildung 4.5b dargestellt.

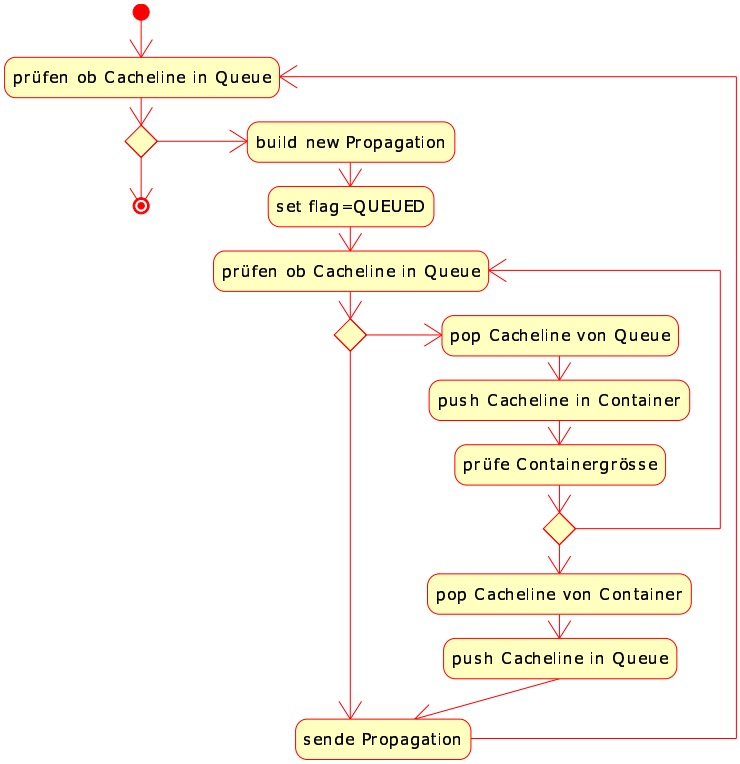


Abbildung 4.5b: queued Propagation

Bis hierhin ist die Propagation innerhalb eines Clusters ausreichend implementiert. Für die Erfüllung der Anforderungen muss hier über die Clustergrenzen hinweg propagiert werden. Alle Komponenten, die bisher implementiert wurden, arbeiten nur clusterweit, da ihre Funktionen nur für den Cluster selbst relevant sind. Der Propagator dagegen ist für die clusterübergreifende Verteilung verantwortlich. Bisher arbeitet er aber mit CellBroadcast, und Propagationen werden nicht erneut propagiert. Gatewayknoten, also Knoten, die in mehr als einem Cluster Mitglied sind, verbreiten Änderungen aus einem Cluster in andere. Allerdings tun sie das nur mit Änderungen, die lokal vorgenommen wurden. Gatewayknoten können aber auch nicht einfach empfangene Propagationen erneut verbreiten, sonst entstehen unendliche Propagationsprozesse. Propagationen müssen mit eindeutigen Sequenznummern ausgestattet werden. Propagationen, deren Sequenznummer nicht größer als die der letzten Propagation ist, werden dann ignoriert. Da eine einzelne, global eindeutige Sequenznummer nicht realisierbar ist, führt jeder Knoten seine eigene Sequenznummer. Zusammen mit der StationID des Ursprungsknoten (**origin**) erhält man einen eindeutigen Identifikator. Eine Propagation ist dann wie in Abbildung 4.5c zu sehen aufgebaut.

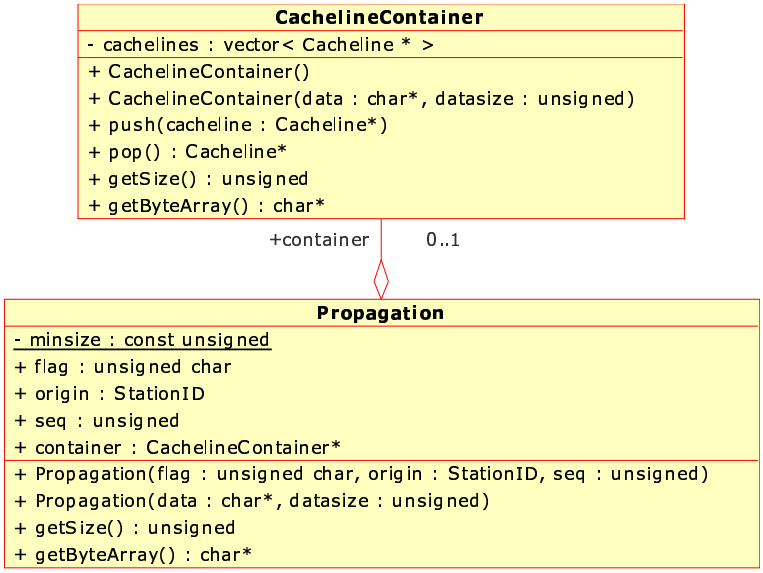


Abbildung 4.5c: Klassendiagramm Propagation

Gatewayknoten können nun erneut propagieren, ohne dass Zyklen entstehen. Cachelines diffundieren über Clustergrenzen hinweg. In Abbildung 4.5c ist das Attribut **flag** zu sehen. Ebenso wie bei der Propagation lokaler Änderungen gibt es auch bei der Repropagation die Unterschiede in der Art der Verarbeitung. Propagationen können direkt erneut versendet werden oder landen erstmal in einer weiteren Queue, aus der sie dann verzögert versendet werden. Zur Unterscheidung dient flag. Ist flag auf **PSCACHEPROPAGATOR_DIRECT** gesetzt, wird dieses Propagationspaket auch sofort wieder versendet. Dies ist der Fall, wenn nur eine Cacheline in diesem Paket direkt propagiert wurde. In allen anderen Fällen kann man von Änderungen durch Hits ausgehen. Flag ist dann **PSCACHEPROPAGATOR_QUEUED**. Die Hitpakete werden nur in periodischen Abständen versendet. Sie diffundieren somit verzögert von Cluster zu Cluster. Eine Analyse der Verbreitungsgeschwindigkeit wurde bereits in Kapitel 3.7.2 durchgeführt. Die periodischen Aktivitäten werden über GEA implementiert. Der Propagator hat somit als Basisklasse neben Module auch die GEA-Klasse **MultiShootTimerEvent**. Sie repräsentiert ein periodisches Ereignis. Bei der Initialisierung wird eine Periode angegeben, in der das Ereignis aktiviert wird. In Abbildung 4.5d ist das Klassendiagramm des Propagators zu sehen.

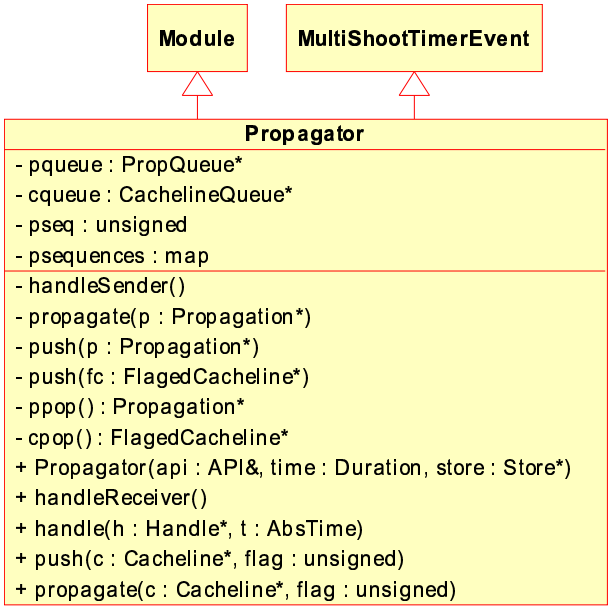


Abbildung 4.5d: Klassendiagramm Propagator

Über die Methode **push()** werden dem Propagator Cachelines übergeben, die über die Queue propagiert werden sollen. Direkte Propagationen werden mit der Methode **propagate()** ausgelöst. Mit **handle()** werden die periodischen Propagationen aus der Queue gehandhabt.

4.6 Cacheanfragen/Querier

Für die Applikationen ist das System völlig transparent. Es ist nichts von der Verteilung bekannt. Applikationen stellen somit Anfragen immer an die lokale Instanz. Ist diese für das entsprechende Bucket zuständig, kann die Anfrage sofort ausgeführt werden. Über eine Callbackfunktion wird das Ergebnis an die Applikation zurück geliefert.

Ist die lokale Instanz nicht zuständig, müssen die anderen Knoten des Clusters befragt werden. Hier kommt der **Querier** zum Einsatz. Die Anfragen, die vom Querier verarbeitet werden, können nur asynchron vom Cluster beantwortet werden. Dies resultiert aus der ereignisbasierten Struktur des Systems. Alle Anfragen müssen somit vom Querier gespeichert werden. Dies geschieht als Instanz der Klasse **Query**. Abbildung 4.6 zeigt das zugehörige Klassendiagramm. Die Abfragezeit muss durch ein Time-out begrenzt sein. Andernfalls wartet die anfragende Applikation unendlich auf ein Ergebnis. Wie auch schon beim Syncer wird GEA genutzt, um das Time-out zu implementieren. Deshalb ist die Basisklasse SingleShootTimerEvent. Der Wert für den Time-out ist statisch mit PSCACHEQUERY_TIMEOUT in „Query.h“ vorgegeben.

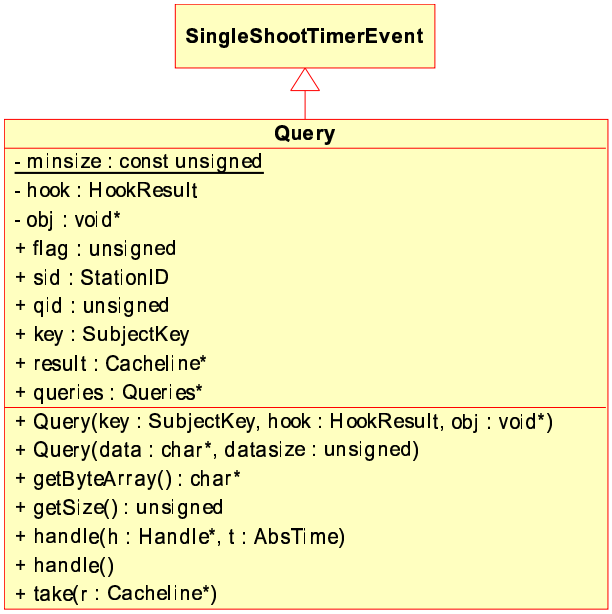


Abbildung 4.6: Klassendiagramm Query

Zur Identifikation der Anfragen dient wie bei den Synchronisationen die Kombination aus StationID und einer Sequenznummer. Letztere wird vom Querier zugewiesen. Es entstehen eindeutig unterscheidbare Anfragen. Der Querier ist eine sehr simple Komponente des Systems. Es sorgt lediglich für das Management der versendeten Anfragen und bearbeitet empfangene Anfragen. Das Klassendiagramm der Klasse **Querier** ist in Abbildung 4.6a zu sehen.

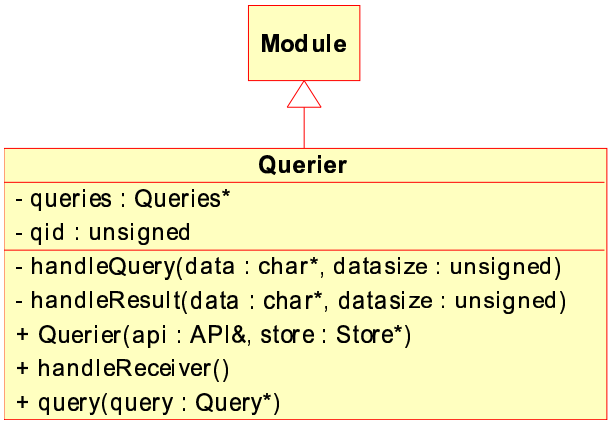


Abbildung 4.6a: Klassendiagramm Querier

Die beiden Methoden **handleQuery()** und **handleResult()** sind für die Verarbeitung der Kommunikationsereignisse zuständig. Anfragen anderer Clustermitglieder werden von handleQuery() verarbeitet und eventuell beantwortet. Die versendeten Ergebnisse der Anfragen werden mit handleResult() verarbeitet. Existiert in **queries** eine pasende Anfrage, wird die empfangene Cacheline mit take() der Anfrage hinzugefügt. Über die Methode **query()** werden die Anfragen an andere Mitglieder dem Querier vom Cache übermittelt.

4.7 SizeGuard

Das SizeGuard-Modul des Cachesystems wartet den Store. Dazu wird die Methode clean() des Store in regelmäßigen Abständen aufgerufen. Diese Aufgabe wird durch die Klasse SizeGuard erledigt. Sie ist wie der Propagator eine Subklasse von MultiShootTimerEvent. Abbildung 4.7 zeigt das Klassendiagramm von SizeGuard.

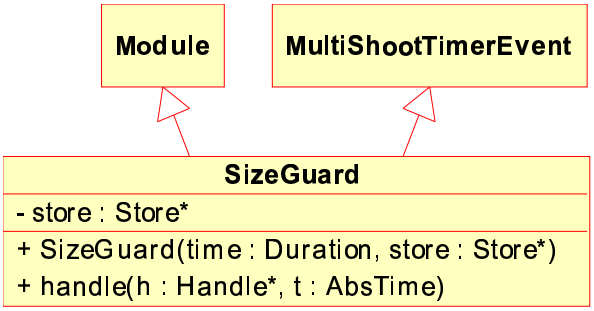


Abbildung 4.7: Klassendiagramm SizeGuard

4.8 Zusammenfassung

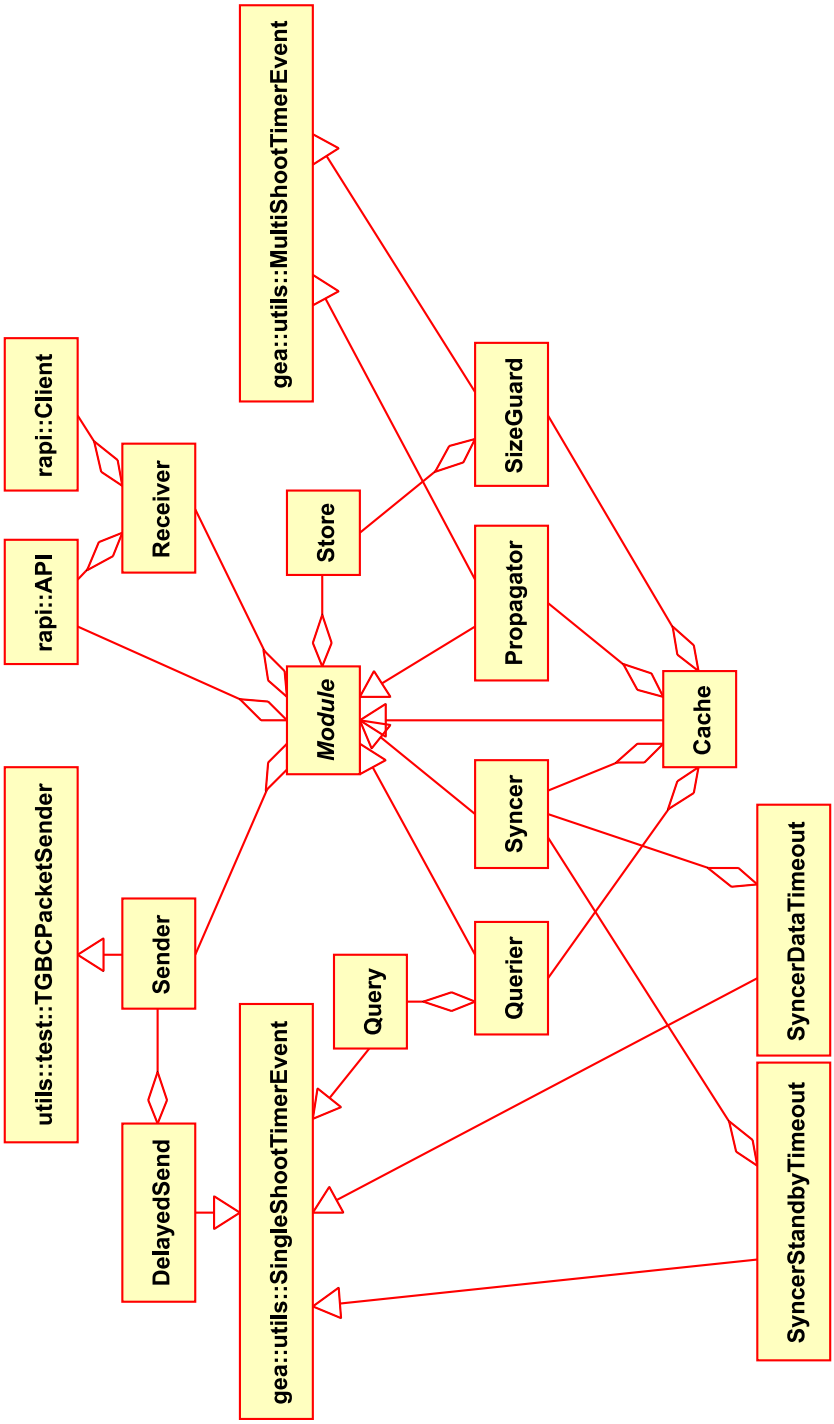


Abbildung 4.8: vereinfachtes Klassendiagramm vollständig

5 Fazit

In dieser Diplomarbeit wurde ein verteiltes Cachesystem für die „Publisher/Subscriber Middleware“ entwickelt. Es integriert sich in die Umgebung der Middleware mit der Verwendung von GEA. Dabei ist das Cachesystem nicht für eine spezielle Anwendung zugeschnitten. Komponenten der Middleware sowie Applikationen, die auf der Middleware basieren, können Informationen in dem einheitlichen Cachesystem zwischenspeichern. Damit sich unterschiedliche Nutzer des Caches nicht gegenseitig die Daten zerstören können, ist dafür eine Aufteilung des Schlüsselraumes notwendig. Anders als bei einem lokalen Cache verbreiten sich die Daten des Cachesystems durch das gesamte Netzwerk. Als Folge dessen können Informationen, die einmal gesammelt wurden, von allen Knoten des Netzes wiederverwendet werden.

Die Middleware stellt Echtzeitkommunikation zur Verfügung. Das Cachesystem beeinflusst die Verfügbarkeit dieser Kommunikationen nicht. Die Kommunikation, die für das Cachesystem benötigt wird, wird vollständig mit Best-Effort-Traffic abgedeckt. Die Middleware versendet Best-Effort-Traffic nur, wenn die verfügbare Bandbreite momentan nicht für Echtzeitkommunikation benötigt wird. Das Cachesystem lebt somit von den Überresten der Bandbreite. Trotzdem erreicht das System Kommunikationseffektivitäten von 90% bis teilweise fast 100%.

Das Netzwerk, das durch die Middleware bereitgestellt wird, ist geclustert. Ein Cluster stellt für das Cachesystem eine datenunabhängige Cacheeinheit dar. Innerhalb eines Clusters sind die Daten disjunkt verteilt. Dabei werden die momentanen Auslastungen innerhalb des zu verteilenden Bereiches an Daten berücksichtigt. Die disjunkten Teile können somit aus unterschiedliche großen Schlüsselintervallen bestehen. Durch diese Auslastungsberücksichtigung ist eine faire Verteilung der Daten gewährleistet. Die Verteilung ist aber auch redundant aufgebaut, so dass der Dynamik des geclusterten Ad-Hoc Netzwerkes der Middleware Rechnung getragen wird. Das Cachesystem passt seine Verteilung den sich ändernden Konstellationen eines Clusters an.

Das Cachesystem ist fehlertolerant aufgebaut. Durch den Verlust von Paketen kann das System nicht in einen undefinierten Zustand übergehen. Differenzen in den Datenbeständen werden generell akzeptiert. Wobei diese nur dem Maß an Fehlerübertragungen entsprechen können. Durch die Propagation aller Änderungen der Daten ist eine kontinuierliche Abgleichung der Daten gegeben. Einmal verlorengegangene Daten werden bei späteren Übertragungen von Änderungen erneut verbreitet.

Die Alterung und Ersetzung von Daten wird auf jedem Knoten einzeln behandelt. Hier kommt Least-Recently-Used für die Ersetzung zum Einsatz. Für jedes Datum des Caches wird ein maximales Alter definiert. Wird dieses erreicht, löscht der Cache das Datum selbstständig.

6 Ausblick

Die „Publisher/Subscriber Middleware“ befand sich zum Zeitpunkt dieser Diplomarbeit noch in der Entwicklung. Eine intensive Testung des Cachesystems konnte somit noch nicht durchgeführt werden. Die zu erwartenden Ergebnisse der stochastischen Analyse aus Kapitel 3.7 müssen sich noch einer experiementellen Überprüfung stellen. Dabei müssen einige Parameter für Time-outs und Übertragungswiederholungen angepasst werden, um das Cachesystem zu optimieren.

Primäre Einsatzgebiete des Cachesystems sind der Namensdienst sowie das Routingprotokoll der Middleware. Der Namensdienst hält Informationen über Publisher und Subscriber und deren Inhalte. Publisher und Subscriber eines Knotens tragen dort ein, welche Inhalte sie publizieren bzw. empfangen möchten. Diese Informationen werden vom Routing gebraucht, um Pakete der Publisher zu den Knoten zu senden auf denen ein Subscriber die Inhalte empfangen möchte. Die Informationen des Namensdienstes sind somit grundlegend für die Publisher/Subscriber-Kommunikation der Middleware. Bisher können diese Informationen nur durch Flutungen reaktiv ermittelt werden. Das Cachesystem bietet nun eine Möglichkeit, diese Informationen proaktiv im gesamten Netz zu verteilen. Es ist zu erwarten, dass die Häufigkeit der Flutungen dadurch stark reduziert wird. Als Folge dessen wird die Größe des Netzes, in dem die Middleware arbeiten kann gößer sein.

Die Einsatzfähigkeit des Cachesystems wird sich in den zukünftigen Entwicklungen der Arbeitsgruppe „Echtzeitsysteme und Kommunikation“ zeigen.

7 Literaturverzeichnis

[Milanovic] N. Milanovic u.a., Ad hoc networks and the wireless internet, 2003

[HJ2001] A. Heinrich, K. Jobmann, Kommunikationsanforderungen ubiquärer Endgeräte, 2001

[Cardei] M. Cardei u.a., Energy Efficient Approaches in Wireless Networking, 2004

[CMANET] , Clustering of Mobile Ad-Hoc Networks to Provide QoS Guarantees, 2005

[CDK2002] George Coulouris, Jean Dollimore, Tim Kindberg, Verteilte Systeme Konzepte und Design, 2002

[EUGSTER] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, Anne-Marie Kermarrec, The many Faces of Publish/Subscribe, 2003

[UML2] Patrick Grässle, Henriette Baumann, Philippe Baumann, UML 2.0 projektorientiert, 2004

[GEA] Andre Herms, Daniel Mahrenholz, Developing and Deploying Event-Driven Protocols with GEA, 2004

[POSIX] The Open Group, Portable Operating System Interface, 2004

[PSMW] , Eine Publisher/Subscriber-basierte Middleware mit Dienstgüte-Garantien zur Unterstützung kooperativer Anwendungen,

[PSMWDFG] Prof. Dr. Edgar Nett, Eine Publisher/Subscriber-basierte Middleware mit Dienstgüte-Garantien zur Unterstützung kooperativer Anwendungen,

[802.11] IEEE, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 1999

[OTCL] , OTcl,

[NSTUT] Padmaparna Haldar, Xuan Chen, Ns Tutorial 2002, 2002

[NSWL] Padma Haldar, Wireless world in NS,

[Bengel] Günther Bengel, Verteilte Systeme: Client-Server-Computing für Studenten und Praktiker, 2002

[Karger97] David Karger, Eric Lehrman, Tom Leighton, Matthew Levine, Daniel Lewin, Rina Panigrahy, Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web, 1997

[HD2] Felix Hausdorff, Grundzüge der Mengenlehre, 1914

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Thomas Draband