Humboldt University Berlin
Computer Science Department
Systems Architecture Group

Rudower Chaussee 25
D-12489 Berlin-Adlershof
Germany

Phone: +49 30 2093-3400
Fax: +40 30 2093-3112
http://sar.informatik.hu-berlin.de

Precaching auf mobilen Geräten

HU Berlin Public Report
SAR-PR-2008-04

Februar 2008

Author(s):
Sebastian Ehrich

Humboldt-Universität zu Berlin
Institut für Informatik
Lehrstuhl für Systemarchitektur

Studienarbeit zum Thema

Precaching auf mobilen Geräten

Sebastian Ehrich

9. Februar 2008

Betreuer: Dr. Wolf Müller
Gutachter: Prof. Dr. Jens-Peter Redlich

1 Einleitung

Mobile Geräte wie Smartphones oder Handheld Computer werden im Gegensatz zum Desktop PC in einer sehr dynamischen Umgebung eingesetzt. Insbesondere existiert bei stationären PC meist eine feste Netzwerk- oder Internetverbindung, während dies bei mobilen Geräten nicht der Fall ist, ob nun wegen wechselnder Signalstärke und -qualität oder hohen Übertragungskosten. Trotz dieses Umstandes, erwarten die Anwender dieser Geräte verlässlichen Zugriff auf Dienste und Daten im Netzwerk, wie sie es von einem stationären Computer gewohnt sind. Um diese Anforderungen auch bei nicht permanenter Netzwerkverbindung zu gewährleisten, ist es notwendig Dienste und Daten aus dem Netzwerk so auf dem mobilen Endgerät zwischenzuspeichern, dass eine Verbindung zum Netzwerk nur selten oder gar nicht erforderlich ist.

In dieser Arbeit konzentriere ich mich auf das vorausschauende Zwischenspeichern (*Precaching* genannt) von Daten und vernachlässige Dienste. Ich stelle ein Programm vor, welches Mono[1]-Objekte auf dem mobilen Gerät zwischenspeichert und bei bestehender Netzwerkverbindung mit einem Server abgleicht. Optionale Regeln ermöglichen es, auf die Arbeitsweise des Caches Einfluss zu nehmen. In dieser Studienarbeit wird der von mir implementierte Cache *ObjectCache* im Details vorgestellt.

Dazu werde ich eine Einordnung des ObjectCache vornehmen, mich der Architektur der Einsatzumgebung des Caches und der des ObjectCache selbst widmen. Es erfolgt eine genauere Erläuterung des regelbasierten Cachings und die Kommunikation zur Aktualisierung der Daten des Caches wird genauer erläutert. Mögliche Erweiterung werden skizziert und eine Übersicht über ähnliche Anwendungen und die Anwendung des ObjectCache selbst wird gegeben.

2 Einordnung

Der Cache ist Teil einer Architektur, die mobiles Precaching von Daten unter Wahrung der Vertraulichkeit der Daten gewährleisten soll. Um dem Nutzer stets nur für ihn relevante Daten zu übermitteln und die übertragene und auf den mobilen Endgeräten zwischengespeicherte Datenmenge so klein wie möglich zu halten, versucht ein Rechner mit aktuellem Datenbestand – *Update-Server* – genannt, stets ein genaues Profil des jeweiligen Nutzers zu erhalten und darauf basierend, möglichst präzise Vorhersagen über dessen voraussichtlich benötigte Daten zu treffen.

Precaching setzt sich aus zwei Komponenten zusammen. Zum einen geht es um das *Caching* von Daten, damit die Informationen auch ohne Netzwerkverbindung zum eigentlichen Speicherort dem Nutzer zur Verfügung stehen. Im Falle des ObjectCache wird sowohl das Lesen als auch das Schreiben der Daten zwischengespeichert. Der zweite Teil besteht aus dem *Prefetching*, welches versucht, Daten, die der Benutzer wahrscheinlich benötigt, schon vor der eigentlichen Anforderung zu laden. Der ObjectCache übernimmt in der Architektur die Aufgabe des Caching, während sich der Update-Server um das Prefetching kümmert.

[1] www.mono-project.com

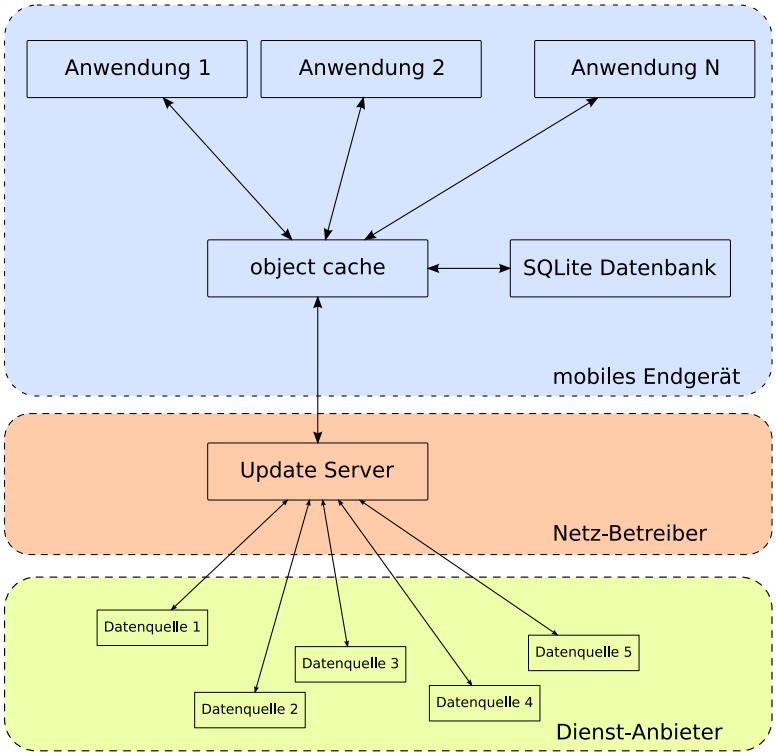

Abbildung 1: Übersicht über die Architektur

Eine Übersicht über die Architektur des Gesamtsystems ist in Abbildung 1 zu finden. Der ObjectCache hat seinen Platz in dieser Architektur auf den mobilen Endgeräten und bietet eine möglichst transparente Schicht zur Speicherung der Daten an. Der ObjectCache ist als Bibliothek implementiert, die mehreren Anwendungen einen geordneten und parallelen Zugriff auf eine Persistenzschicht ermöglicht. Die Plattform auf dem der Cache laufen wird ist OpenMoko[2], ein auf Linux basierendes, offenes Betriebssystem für mobile Geräte, vorallem das FIC Neo1973 Smartphone[3].

3 Anforderungen

Der ObjectCache soll auf mehreren mobilen Plattformen zum Einsatz kommen und zusammen mit dem Update-Server für eine Synchronisierung der Arbeitsumgebung auf den einzelnen Geräten sorgen. Um möglichst Plattform unabhängig zu sein, wird der ObjectCache in Mono implementiert, einer offene Implementation des .NET-Frameworks. Mit Hilfe von Mono ist es möglich Anwendungen zu entwickeln, die auf mehreren Plattformen lauffähig sind, was im Falle dieser Architektur ermöglicht, eine Anwendung auf mehreren mobilen Geräten und insbesondere mehreren Betriebssystem-Umgebungen einzusetzen.

[2] www.openmoko.org
[3] wiki.openmoko.org/wiki/Neo1973

Anwendung werden daher auch als eine weitere Art von Daten behandelt, die zwischen den einzelnen Geräten eines Nutzers synchronisiert werden.

Da auf einem Mobilgerät mehrere Anwendungen gleichzeitig den Cache nutzen können sollen, ist es nötig, die Zugriffe der einzelnen Anwendungen nur auf die von ihnen eingefügten, bzw. die ihnen zugeordneten Daten zu beschränken. Außerdem soll jede Anwendung unabhängig wichtige Parameter des Cachings (z.B. die Gültigkeitsdauer von Objekten) festlegen können. Diese sollen sich gezielt für einzelne Gruppen von Objekten festlegen lassen. Die einzelnen Objekte sollen unabhängig von ihrem Inhalt im Cache gespeichert werden. Das läuft unter Mono darauf hinaus, dass Objekte der allgemeinsten Klasse `object` gespeichert werden müssen. Es ist wichtig, mit dem Objekt auch Informationen über die Objektklasse mitzuführen, damit beim Auslesen aus dem Cache klar ist, welchen Typ ein gespeichertes Objekt hat. Es existieren prinzipiell zwei Zustände – on- und offline, Daher muss es möglich sein, im Betrieb mit Netzwerkverbindung auch direkt einzelne Objekte aus dem aktuellen Datenbestand des Update-Servers zu erhalten. Im Offline-Betrieb sollen Änderungen an den Objekten zunächst lokal zwischengespeichert und bei wieder hergestellter Netzwerkverbindung mit dem Update-Server synchronisiert werden. Um ein Profil der vom jeweiligen Benutzer wahrscheinlich benötigten Daten anzufertigen, ist es notwendig, die Zugriffe und Anfragen, auf Objekte aus dem Object-Cache, zu protokollieren. Die protokollierten Daten werden bei der Synchronisation an den Update-Server übermittelt. Dieser soll mit Hilfe der Informationen über die für den Nutzer interessanten Objekte die zu übermittelnden Daten im Voraus berechnen. Die Implementation des Update-Servers liegt aber außerhalb des Fokus dieser Studienarbeit. Einige eher theoretische Betrachtungen zu diesem Thema finden sich beispielsweise in den Arbeiten [3, 8].

4 Architektur

Der folgende Abschnitt gibt eine Übersicht, sowohl über die Architektur des Gesamtsystems und den Platz des ObjectCache darin, als auch über den eigentlichen Aufbau des ObjectCache selbst.

4.1 Gesamtarchitektur

Wie man in der Architekturübersicht (Abbildung 1) erkennen kann, besteht das Gesamtsystem aus drei Schichten. Eine Schicht bilden die Diensteanbieter, die im Internet verschiedenste Dienste und Daten zur Verfügung stellen und welche durch den Anwender benutzt werden. Dabei sind die Dienste verschiedenartigster Herkunft und unterstehen der Kontrolle der verschiedensten Organisationen und/oder Firmen. Es wird angenommen, dass die Dienste entsprechende Schnittstellen vorsehen, über die der Update-Server mit ihnen kommunizieren kann. Diese Schnittstellen können z.B. explizit Web-Services sein, es ist aber auch denkbar, dass der Update-Server die benötigten Daten aus der HTML-Beschreibung einer Seite des Dienstes extrahiert.

Die Daten dieser Dienste werden von der zweiten Schicht – der des Update-Server – zwischengespeichert. Der Update-Server wird wahrscheinlich oft vom Netz-Anbieter des

Mobilfunknetzes des Anwenders betrieben, da die Netzbetreiber schon jetzt versuchen, mobilen Anwendern durch den Einsatz von Proxy-Server den mobilen Internetzugang zu vereinfachen. Darüber hinaus hat insbesondere der Zugangsanbieter die Möglichkeit zuverlässig zu ermitteln, wann der Nutzer eine Datenverbindung aufbaut und kann dann eine Synchronisation von ObjectCache und Update-Server anstoßen. Der oder die Update-Server bilden einen Fixpunkt in der Architektur, da sämtliche Kommunikation zwischen Diensten und dem Endgerät nur indirekt über diesen oder diese Rechner laufen. In dieser Schicht wird dann auch ein Profil der Daten des Anwenders erstellt und es werden vorausschauend Daten der Dienste abgerufen, um sie dem Endgerät, schon vor einer entsprechenden Benutzeranfrage, zu übermitteln. Die Update-Server haben außerdem die Aufgabe, Daten die der Nutzer auf einem mobilen Gerät erstellt, für die Synchronisation mit anderen mobilen Geräten des Nutzers vorzuhalten.

Die dritte Schicht bilden der ObjectCache und die Anwendungen auf dem mobilen Endgerät. Auf diese Schicht konzentriere ich mich in dieser Arbeit. Hier werden die vom Update-Server ausgewählten und empfangenen Daten im ObjectCache aufbewahrt und auf Anfrage hin den einzelnen Applikationen auf dem mobilen Endgerät zur Verfügung gestellt. Der ObjectCache kümmert sich um die Verwaltung der Daten und bietet den Anwendungen eine Schnittstelle zum Abruf an.

4.2 Architektur des ObjectCache

Die Hauptkomponenten der Architektur des ObjectCache sind in Abbildung 2 zu sehen. Die einzelnen Komponenten sind durch ihre Schnittstellen innerhalb und zu externen Systemen beschrieben. Dadurch ist es leicht möglich, für die einzelnen Komponenten des ObjectCache neue Implementationen zu schreiben und das Verhalten des Cache neu zu definieren. Der ObjectCache besteht im Groben aus vier Komponenten die in den folgenden Abschnitten näher beleuchtet werden.

4.2.1 Modellierung der Cache-Objekte

Das grundlegende Objekt im ObjectCache ist durch die Klasse `CacheObject` modelliert. In dieser werden sämtliche wichtigen Informationen zu einem Objekt im ObjectCache gespeichert. Dies umfasst folgende Daten:

Anwendungs-ID Diese gibt an, zu welcher Anwendung das Objekt gehört.

Objekt-ID Die Objekt-ID ist frei wählbar, sollte aber für zwei unterschiedliche Objekte auch unterschiedlich sein. Der Aufbau der Objekt-IDs ist gänzlich dem Nutzer überlassen. Sinnvoll wäre zum Beispiel ein hierarchischer Aufbau, der die Objekte in verschiedenen Kategorien gliedert. So wäre es denkbar alle Bilder-Objekte mit einer ID nach dem Schema „`picture.IMAGE_ID`“ zu versehen.

Zeitstempel Dieser gibt an, wann das Objekt das letzte Mal aktualisiert wurde. Die Aktualisierung kann durch einen Abgleich mit dem Update-Server, durch ein Editieren oder das Erstellen des Objektes durch den Benutzer, erfolgt sein.

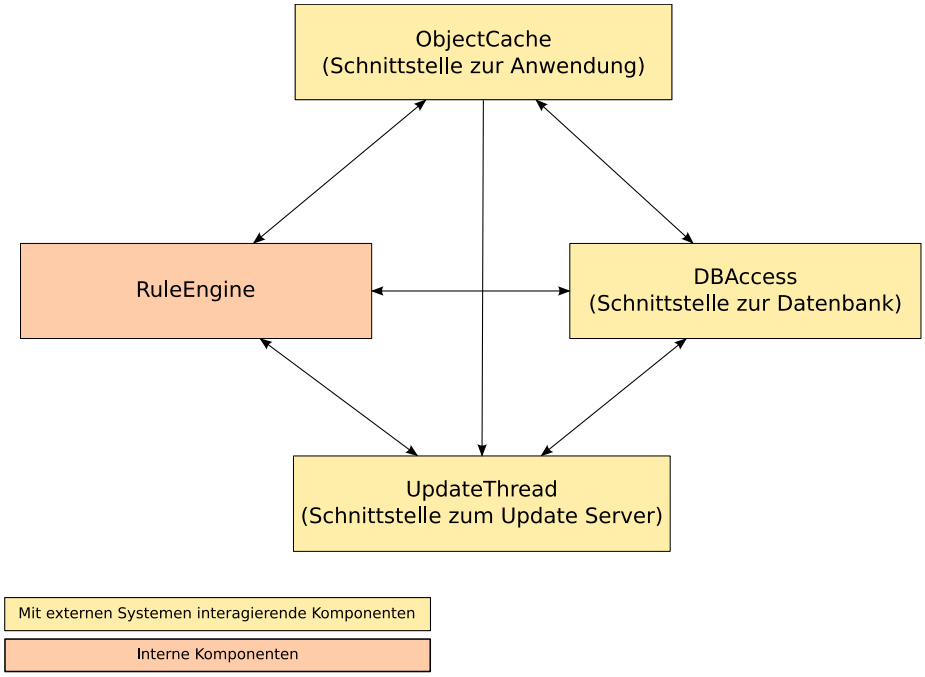

Abbildung 2: Architektur des ObjectCache

Typ Speichert den Typ des Objektes, damit auch ohne Auslesen des Objektes festgestellt werden kann, welchen Typs das gespeicherte Objekt ist. Dies ist besonders wichtig beim Auslesen des Objektes, damit es auf den richtigen Typ gecastet werden kann.

Objekt-Daten Darin wird das eigentliche Objekt in serialisierter Form gespeichert. Das Objekt wird in der jetzigen Implementation als Base64-kodierte Binärdaten abgelegt. Es ist aber auch möglich, bei Benutzung eines entsprechenden Serialisierers, das Objekt z.B. als XML abzulegen und es damit auch möglich zu machen, ohne Deserialisierung einige Objekteigenschaften auszulesen.

4.2.2 Schnittstelle zur Anwendung

Die Schnittstelle zur Anwendung umfasst die in Tabelle 1 aufgeführten Methoden. Diese lassen sich grob in vier Kategorien teilen. Als erstes gibt es Methoden, die sich mit den eigentlichen Daten-Objekten beschäftigen und diese hinzufügen oder auslesen. Die zweite Kategorie besteht aus den Methoden, die mit Regeln arbeiten. Mit Hilfe dieser, können Regeln hinzugefügt, ausgelesen, gelöscht und aktualisiert werden. Über diese Methoden erfolgt die Kommunikation der einzelnen Anwendung zur Regel-Einheit. Eine weitere Art von Methoden beschäftigen sich mit der Aktualisierung der im Cache vorhandenen Daten. Ein Aufruf der einzigen Methoden dieser Kategorie, führt dazu, dass ein neuer UpdateThread gestartet wird, welcher die Aktualisierung vornimmt. Die letzte Kategorie besteht wiederum aus einer Methode, die sich um die Verwaltung der einzelnen

Methodenname	Beschreibung
`Add(ObjectID, Object, Type)`	Fügt das gegebene Objekt unter der gegebene Objekt-ID in den ObjectCache ein und speichert dazu den angegeben Typ.
`Get(ObjectID, OnlineFlag, out Type)`	Liest das Objekt mit der gegeben Objekt-ID aus dem ObjectCache und liefert zusätzlich den für das Objekt gespeicherten Typ. Das OnlineFlag gibt an, ob das Objekt direkt beim Update-Server angefragt werden (Wert `true`) oder aus dem lokalen ObjectCache (Wert `false` stammen soll.
`IdsMatching(Regex)`	Liefert alle Objekt-IDs aus dem ObjectCache zurück, die auf den angegebenen regulären Ausdruck passen.
`AddRule(Rule)`	Fügt die angegebene Regel in den ObjectCache ein.
`DeleteRule(Regex)`	Löscht alle Regeln, die den gegebenen regulären Ausdruck als Selektionsparameter ausweisen, aus dem ObjectCache.
`UpdateRule(Rule)`	Überschreibt eine eventuell vorhandene Regel, mit gleichem Selektionsparameter, mit der gegebenen Regel.
`ReadRule(Regex)`	Liest die resultierende Regel für den angegebenen Selektionsparameter aus.
`UpdateCache()`	Aktualisiert den gesamten Datenbestand des ObjectCache mit Hilfe des Update-Server.

Tabelle 1: Methoden der Anwendungsschnittstelle

Applikationen kümmert. Sie dient dazu den einzelnen Anwendungs-IDs jeweils sinnvolle Namen zuzuordnen. In einer erweiterten Implementation wäre es denkbar, noch weitere Methoden hinzuzufügen, die sicherstellen, dass die Vergabe und Anwendung der Anwendungs-IDs auf eine sichere Art geschieht. Bisher genügt allein die Kenntnis einer Anwendungs-IDs, um die Daten einer anderen Applikation auslesen zu können.

Eine weitere Aufgabe der ObjectCache Komponente ist, neben Speichern und Lesen der Objekte in bzw. aus der Datenbank, mit Hilfe der Regel-Einheit zu prüfen, ob die einzelnen auszuführenden Aktionen, zulässig sind. So wird vor dem Hinzufügen eines Objektes zum Cache jeweils ermittelt, ob ein Objekt mit der gegebenen Objekt-ID überhaupt in den Cache eingefügt werden darf, ob schon ein Objekt mit der gleichen Objekt-ID vorhanden ist und ob dieses dann überschrieben werden darf oder nicht.

4.2.3 Schnittstelle zur Datenbank

Als Datenbank kommt beim ObjectCache SQLite[4] zum Einsatz. Da diese Datenbank-Implementation große Teile der SQL92 Spezifikation[5] und Transaktionen umsetzt, trotzdem relativ wenig Speicher belegt und sich einfach verwenden lässt, eignet sich SQLite sehr gut, um auf einem mobilen Gerät mit begrenzten Ressourcen eingesetzt zu werden. Auch die Verfügbarkeit auf einer großen Anzahl von Plattformen ist ein bedeutender Vorteil.

Wie im Schema der Datenbank in Abbildung 3 zu sehen ist, besteht die Datenbank aus vier Tabellen.

Die Tabelle `Applications` dient dazu, die Relation zwischen Anwendungs-ID und Anwendungsname zu speichern.

Die eigentlichen Objekte werden in der Tabelle `Data` gespeichert. Zu den eigentlichen Objekt-Daten wird noch ihr Objekt-Typ, der Zeitpunkt der letzten Modifikation des Objektes und die Objekt-ID abgelegt. Um die Zuordnung zu einer bestimmten Anwendung zu sichern, wird jedem Datensatz noch die Anwendungs-ID des Programms hinzugefügt, von dem das entsprechende Objekt eingefügt wurde. Weiterhin existiert ein Feld, welches eine Markierung enthält, die angibt, ob das Objekt lokal verändert wurde und die Änderungen somit an den Update-Server übertragen werden müssen.

In der Tabelle `Requests` werden die Anfragen nach Objekten, die von den Anwendungen an den Cache gestellt werden, abgelegt. Für die Speicherung der Anfrage ist es nötig, die Anwendungs-ID abzulegen, um zu wissen, welche Anwendung angefragt hat. Außerdem wird der Zeitpunkt, an dem die Anfrage erfolgte, protokolliert und durch die Speicherung der Objekt-ID wird festgehalten, nach welchem Objekt gefragt wurde. Dabei werden sowohl erfolgreiche, als auch nicht erfolgreiche Anfragen und das Anlegen neuer oder die Änderung existierende Objekte als Anfrage gewertet.

Die vierte und letzte Tabelle `Rules` dient dazu die Regeln, die von den einzelnen Anwendungen definiert werden, zu speichern. Dem bekannten Schema folgend, wird wieder die Anwendungs-ID abgelegt. Es wird der reguläre Ausdruck gespeichert. Über diesen werden die Objekte, auf die die Regeln angewendet werden sollen, ausgewählt.

[4] www.sqlite.org
[5] ISO/IEC 9075:1992

Applications		
applicationID	text	ID der Anwendung
applicationName	text	Name der Anwendung

Data		
applicationID	text	ID der Anwendung
timestamp	integer	Zeitpunkt der letzten Modifikation
objectID	text	ID des Objekts
data	blob	Objekt-Daten
type	text	Typ des Objekts
modified	integer	Modifikations-Markierung

Requests		
applicationID	text	ID der Anwendung
timestamp	integer	Zeitpunkt der letzten Anfrage
objectID	text	ID des angefragten Objekts

Rules		
applicationID	text	ID der Anwendung
objectRegex	text	Regulärer Ausdruck zur Selektion
rule	blob	Regel-Objekt-Daten

Abbildung 3: Schema der Datenbank

Aktion	Beschreibung
None	Keine Aktion
InsertObject	Einfügen eines Objektes
UpdateObject	Aktualisieren eines vorhandenen Objektes
InsertObjectDuplicate	Hinzufügen eines weiteren Objektes mit einer schon vorhandenen Objekt-ID
ReadObject	Lesen eines Objektes
InsertRequest	Einfügen einer Anfrage
UpdateRequest	Aktualisieren einer vorhandenen Abfrage
InsertRequestDuplicate	Einfügen einer weiteren Abfrage mit einer schon vorhandenen Objekt-ID
ReadRequest	Lesen einer Abfrage
SendRequest	Übermitteln einer Anfrage an den Update-Server

Tabelle 2: Mögliche Aktionen auf Objekten

Außerdem dient der Ausdruck auch dazu, die einzelnen Regeln zu identifizieren. Die eigentliche Regel wird als serialisiertes Mono-Objekt im Attribute `rule` abgelegt.

Die Schnittstelle `DbAccess`, über die die Datenbank angesprochen wird, sieht für jede Art von Objekten (Anwendung, Regeln, Anfragen, Daten-Objekte) Methoden vor, um diese in die Datenbank einzufügen, zu lesen und zu löschen. Für Anwendungen, Anfragen und Daten-Objekte existieren Methoden, um festzustellen, ob diese schon gespeichert sind und Anfragen und Daten-Objekte können zusätzlich durch die Datenbankschnittstelle aktualisiert werden.

4.2.4 Schnittstelle zur Regel-Einheit

Die Regeleinheit verwaltet die einzelnen Regeln, die vom Benutzer für verschiedene Gruppen von Objekten festgelegt werden. Die Regeln werden auch in der Datenbank abgelegt. Es ist jedoch zu beachten, dass für jeden regulären Ausdruck der festlegt, auf welche Menge von Objekten die Regel angewendet werden soll – und daher Selektionsparameter genannt wird –, immer nur eine Regel existiert. Erstellt man mehrere Regel-Definitionen für den selben Selektionsparameter (dazu siehe Abschnitt 5.1), werden die einzelnen Definitionen zu einer Regel verschmolzen. Wie dies geschieht wird in Unterabschnitt 5.1 näher beschrieben.

Die Schnittstelle bietet mit den Methoden `AddRule`, `DeleteRule`, `UpdateRule` und `ReadRule` Methoden zum Hinzufügen, Löschen, Aktualisieren und Lesen von Regeln. Eine weitere Methode `ClearRules` löscht alle vorhandenen Regel und setzt die Regeleinheit somit zurück. Der wichtigste Teil dieser Schnittstelle ist aber die Methode `CanDo`. Diese bestimmt zu einem gegebenen Objekt und einer gegebenen Aktion, ob die Aktion auf das Objekt angewandt werden darf. Eine Übersicht über mögliche Aktionen ist in Tabelle 2 zu sehen. Die Aktionen werden in einer Aufzählung `Actions` in der Schnittstellenbeschreibung definiert. Zur Bestimmung, ob eine bestimmte Aktion auf das gegebene

Objekt angewendet werden darf, werden aus den vorhandenen Regeln die ermittelt, deren Selektionsparameter auf die ID des Objekts passt. Diese Menge an Regeln wird dann zu einer Regel verschmolzen, die letztendlich über ihre Optionen angibt, ob die gewählte Aktion durchgeführt werden darf.

4.2.5 Schnittstelle zum Update-Server

Der Update-Server versorgt das mobile Endgerät mit aktuellen Informationen und ermittelt die Daten, die wahrscheinlich in der nahen Zukunft vom Benutzer angefordert werden. Die Kommunikation mit dem Update-Server wird im ObjectCache in einem separaten Thread abgewickelt. Dessen Schnittstelle wird in der Klasse `UpdateThread` festgelegt. Die Schnittstelle besteht aus zwei Methoden. Die Methode `run` stößt eine vollständige Aktualisierung aller Objekte im ObjectCache an und übermittelt auch geänderte Daten an den Update-Server. Die zweite Schnittstellen-Methode `fetchImmediate` findet Verwendung, wenn gezielt ein Objekt angefragt wird, dass direkt vom Update-Server geholt werden soll. Dabei werden keine anderen Daten zwischen dem ObjectCache und dem Update-Server abgeglichen. Der genaue Ablauf der Kommunikation mit dem Update-Server ist in Abschnitt 6 beschrieben.

5 Regelbasiertes Caching

Der ObjectCache ist darauf ausgelegt, möglichst jedes serialisierbare Mono-Objekt zwischenspeichern zu können. Es ist aber nicht davon auszugehen, dass für jedes Objekt auch die gleichen Bedingungen bezüglich des Speicherns gelten sollen. So ist zum Beispiel denkbar, dass einige Objekte eher konstant sind und daher nicht häufig aktualisiert werden müssen oder enggültig abgelaufen sind und daher aus dem gesamten System entfernt werden können. Andere zwischengespeicherte Objekte, die sich häufiger ändernde Sachverhalte modellieren, sollen dagegen schneller ihre Gültigkeit verlieren. Auch ist es vorstellbar, dass Anwendungen Objekte als privat deklarieren und diese nicht an den Update Server übermitteln möchten.

Für diese Anforderungen bietet der ObjectCache die Möglichkeit, anhand der Objekt-ID Regeln zu definieren. Diese bestimmen dann, wie lange ein oder mehrere Objekte zwischengespeichert werden, ob sie überhaupt in den Cache übernommen werden sollen oder ob sie an den Update-Server bei der Aktualisierung übermittelt werden können. Auch kann festgelegt werden, ob Zugriffe auf dieses Objekte protokolliert werden sollen.

Dazu unterstützt der Objekt-Cache eine Reihe von Optionen für Regeln, die in Tabelle 3 angegeben sind. Eine Regel wird im ObjectCache durch die Klasse `CacheRule` modelliert. Diese besteht aus zwei Teilen: dem Selektionsparameter und einer Menge von Optionen mit zugehörigen Werten. Der Selektionsparameter ist ein regulärer Ausdruck, der benutzt wird, um zu bestimmen, für welche Menge an Objekten – jeweils identifiziert durch ihre Objekt-ID – die Regeln gültig sein sollen. Dazu wird der reguläre Ausdruck der Regel mit der Objekt-ID verglichen. Die Menge von Optionen und dazugehörigen Werten gibt an, welche Optionen auf die angesprochene Menge von Objekten angewendet werden soll. Als Optionsname und -wert können beliebige Zeichenketten verwendet

Name: sendRequest		
Wertebereich	Standardwert	Beschreibung
$\{0, 1\}$	1	Gibt an, ob die entsprechende Anfrage an den Update-Server geschickt werden soll. Bei 0 wird die Anfrage nicht gesendet, bei 1 schon.
Name: objectValidFor		
Wertebereich	Standardwert	Beschreibung
int	0	Gibt die Zeit in Minuten an, für die das Objekt zwischengespeichert werden soll. Werte < 0 bedeuten, dass das Objekt gar nicht zwischengespeichert werden soll, 0 bedeutet, dass es nie verfällt und Werte > 0 geben eine Zeit in Minuten an.
Name: requestValidFor		
Wertebereich	Standardwert	Beschreibung
int	0	Gibt die Zeit in Minuten an, für die die Anfrage zwischengespeichert werden soll. Werte < 0 bedeuten, dass die Anfrage gar nicht zwischengespeichert werden soll, 0 bedeutet, dass es nie verfällt und Werte > 0 geben eine Zeit in Minuten an.
Name: overwriteObjectDuplicates		
Wertebereich	Standardwert	Beschreibung
$\{0, 1\}$	1	Gibt an, ob bei einem schon in der Datenbank vorhandenen Objekt mit gleicher ID, das vorhandene Objekt überschrieben oder ein neues hinzugefügt werden soll. Bei 0 wird hinzugefügt, bei 1 überschrieben.
Name: overwriteRequestDuplicates		
Wertebereich	Standardwert	Beschreibung
$\{0, 1\}$	1	Gibt an, ob bei einer schon in der Datenbank vorhandenen Anfrage mit gleicher ID, die vorhandene Anfrage überschrieben oder eine neue hinzugefügt werden soll. Bei 0 wird hinzugefügt, bei 1 überschrieben.

Tabelle 3: Optionen für Regeln

werden, was eine spätere Erweiterung der Menge der Optionen vereinfacht.

5.1 Verwaltung der Regeln

Die Verwaltung der gespeicherten Regeln erfolgt in der Klasse `CacheRuleList`. Für jeden regulären Ausdruck existiert genau eine Regel, die die Optionen definiert. Probleme ergeben sich, wenn eine Regel schon vorhanden ist und eine weitere Regel mit dem gleichen Selektionsparameter hinzugefügt werden soll. Dann stellt sich die Frage, ob die vorhandene Regel erweitert, aktualisiert oder komplett ersetzt werden soll. Bei der Erweiterung der vorhandenen Regel werden nur neue Optionen, die in der vorhandenen Regel noch nicht gesetzt sind, hinzugefügt. Werte vorhandener Optionen werden nicht überschrieben. Dieses Verhalten wird durch die Methode `Extend` der Klasse `CacheRule` implementiert. Soll eine vorhandene Regel aktualisiert werden, werden Werte schon vorhandener Optionen überschrieben und neue Optionen hinzugefügt. Optionen und deren Werte, die in der schon vorhandenen Regel existieren, in der neuen Regel aber nicht, werden jedoch beibehalten. Dies wird durch die Methode `Update` implementiert. Soll eine Regel komplett ersetzt werden, werden alle Optionen und Werte der alte Regel verworfen und die Optionen und Werte der neuen Regel kopiert. Dies wird in der Methode `Replace` der Klasse `CacheRule` umgesetzt.

Die Klasse `CacheRuleList` nutzt diese drei Methoden dann um folgendes Verhalten zu implementieren:

- Soll durch einen Aufruf der Methode `CacheRuleList.Add` eine neue Regel eingefügt werden und existiert schon eine Regel mit dem gleichen regulären Ausdruck als Selektionsparameter, wird die vorhandene Regel *aktualisiert*.

- Soll durch den Aufruf von `CacheRuleList.Extend` eine vorhandene Regel *erweitert* werden, wird die gleichnamige Methode der Klasse `CacheRule` aufgerufen. Ist eine Regel mit entsprechendem Selektionsparameter noch nicht vorhanden, wird die Regel wie oben beschrieben hinzugefügt.

- Die Methode `CacheRuleList.Replace` implementiert das *Ersetzen* einer Regel, ruft die gleichnamige Methode in `CacheRule` auf und fügt bei Nichtvorhandensein einer entsprechenden Regel die neue Regel mittels `CacheRuleList.Add` hinzu.

Weithin existieren in der Klasse `CacheRuleList` die Methoden `ReadRule`, `Remove` und `ReadMatching`. Die Methode `ReadRule` liefert zu einem bestimmten regulären Ausdruck die Regel zurück, die diesen als Selektionsparameter besitzt. Die Methode `Remove` entfernt eine Regel komplett. Die wichtigste und komplexeste Methode der Klasse stellt aber `ReadMatching` dar. Diese Methode soll zu einer gegebenen Objekt-ID eine kumulierte Regel liefern, die die Optionen aus genau den Regeln enthält, deren Selektionsparameter, d.h. deren reguläre Ausdrücke, auf die gegebene Objekt-ID passen.

Dazu wird zunächst der Selektionsparameter jeder Regel mit der Objekt-ID verglichen. Passt er auf die Objekt-ID, wird die Regel in einer Liste der passenden Regeln,

zusammen mit den Angaben zur Übereinstimmung von Objekt-ID und regulärem Ausdruck, gespeichert. Sind alle passenden Regeln ermittelt, ist es notwendig zu bestimmen welche Regeln höhere Priorität besitzen als andere. Dazu ein Beispiel.

Es ist denkbar, dass eine allgemeine Regel für alle Objekte existiert, die festlegt wie lange ein Objekt höchstens zwischengespeichert werden soll. Für diese Regel würde der reguläre Ausdruck „`.*`“ verwendet. Nehmen wir nun an, dass alle Listen mit einer Objekt-ID mit Prefix `lists.` gespeichert werden. Soll nun für alle Listen, eine andere Speicherzeit gesetzt werden, wird dafür eine Regel mit dem regulären Ausdruck „`lists.*`“ als Selektionsparameter erstellt. Die Regel mit „`lists.*`“ sollte eine höhere Priorität bekommen, als die Regel mit „`.*`“ als Selektionsparameter und somit deren Optionen und Werte überschreiben. Es kann aber nicht allgemein vom ObjectCache entschieden werden, welche Regel höher zu priorisieren ist.

Um dieses Problem zu lösen, muss der Nutzer des ObjectCache jeweils zwei Vergleiche implementieren und dem ObjectCache als Objekte (des Typs `Comparer`) übergeben. Diese heißen `MatchCompararator` und `RegexComparator`. Der `MatchComparator` vergleicht die Übereinstimmungen von Objekt-ID und zwei verschiedenen regulären Ausdrücken, um zu bestimmen, welcher Ausdruck „besser“ auf die Objekt-ID passt. Für den Verglich gibt es eine Standard-Implementation, die den regulären Ausdruck mit einer größeren Anzahl von Übereinstimmungen bevorzugt. Sehr präzise ist dieses Vorgehen aber nicht. So ergibt zum Beispiel der Ausdruck „`a`“ gegenüber dem Ausdruck „`abrakad`“ beim Vergleich mit der ID „`abrakadabra`“ viel mehr Übereinstimmungen, trotzdem ist es wahrscheinlicher, dass dem zweiten Ausdruck eine höhere Priorität eingeräumt werden soll.

Beim Vergleichen der regulären Ausdrücke anhand ihrer Übereinstimmungen mit der Objekt-ID kann es vorkommen, dass zwei reguläre Ausdrücken genau die selben Übereinstimmungen aufweisen. Um zwischen diesen eine Rangfolge festlegen zu können, werden diese anhand der regulären Ausdrücke selbst verglichen.

Dazu muss dem ObjectCache ein zweiter Vergleich – der `RegexComparator` – übergeben werden. In diesem werden zwei reguläre Ausdrücke verglichen und bestimmt, welcher die höhere Priorität hat. Standardmäßig wird dieser Vergleich über die Länge implementiert, wobei der längere Ausdruck eine höhere Priorität zugesprochen bekommt. Dies fußt auf der Annahme, dass längere reguläre Ausdrücke selektiver sind. Dass dies aber nicht allgemeingültig ist, zeigt folgendes Beispiel. So würde, wenn man die Menge der IDs mit einer Menge deutscher Sätze annimmt, der wesentlich kürzere Ausdruck „`.*y.*`“ wesentlich weniger Treffer liefern und somit selektiver sein, als der längere Ausdruck „`.*(der|die|das).*`“. Allgemein ist es aber, wie in [9] gezeigt wurde, schwierig die Gleichheit von zwei regulären Sprachen (in diesem Fall gegeben durch zwei reguläre Ausdrücke) zu bestimmen . Trotzdem ist das Äquivalenzproblem für reguläre Sprachen entscheidbar, ein Algorithmus dafür ist in [1] zu finden. Für den konkreten Einsatz im ObjectCache wäre dieser Algorithmus aber zu ineffizient.

Nachdem beide gegebene Vergleiche angewendet wurde, liegen die Regeln nach ihrer Priorität sortiert vor. Es sollte stets gelten, dass wenn R_1 höher priorisiert ist als R_2, dann $R_1 \leq R_2$, bei allen Vergleichen und $R_1 < R_2$ bei mindestens einem Vergleich. Die Regeln werden jetzt beginnend mit der am höchsten priorisierten Regel zu einer neuen Regel

verschmolzen. Die vorhandene Regel wird mit der am höchsten priorisierten Regel aus der Liste des passenden Regeln *erweitert*. Das heißt, es werden nur neue Optionen und Werte hinzugefügt, alle schon vorhandenen Optionen und Werte bleiben bestehen. Ist eine Regel verschmolzen worden, wird sie aus der Liste der passenden Regeln entfernt. Solange noch Regeln in der Liste sind, wird dieser Vorgang wiederholt. Nach dem Verschmelzen aller passenden Regeln entsteht eine Menge von Optionen, die, wieder in ein Regel-Objekt verpackt, das Ergebnis der Methode darstellt.

Werden keine passenden Regeln gefunden, wird als Ergebnis die allgemeinste Regel, mit dem regulären Ausdruck „`.*`“ und einer leeren Menge von Optionen, zurückgegeben.

6 Kommunikation zwischen Endgerät und Server

In der jetzigen Version der Gesamtarchitektur ist nur vorgesehen, dass der Update-Server eine Reihe von Diensten anbietet, die der mobile Nutzer anfragen kann. Später soll es auch möglich sein, dass der Nutzer selbst angeben kann, welchen Dienst er nutzen möchte, ohne dass der Server diesen Dienst per se kennen muss. In der in dieser Studienarbeit vorgestellten Version fragt der ObjectCache nur an, ob der Server eine bestimmte Anwendung unterstützt. Später soll der mobile Cache direkt einen neuen Dienst beim Update-Server registrieren können. Dazu muss der Dienst eine Schnittstelle exportieren, über die der Server mit ihm kommunizieren kann. Der mobile Nutzer soll alle nötigen Informationen und Parameter bereit stellen, mit denen der Server den Dienst nutzen kann.

Der Abgleich zwischen Server und Cache gleicht einer Synchronisation. Der Client sendet Informationen über Anfragen nach Objekten und die Daten/Änderungen der geänderten Objekte. Der Server schickt darauf hin die Objekte zum mobilen Gerät, die von diesem wahrscheinlich benötigt werden und sich auf der Seite des Update-Servers geändert haben.

6.1 Beschreibung des Protokolls

In der jetzigen Implementation erfolgt die Kommunikation mit dem Update-Server über ein sehr rudimentäres Protokoll. Dieses soll als Beispiel dienen und ist für einen praktischen Einsatz eher ungeeignet. Es ist in Abbildung 4 dargestellt.

Es sind zwei Szenarien zu unterscheiden. Einerseits kann es sich um ein Update sämtlicher im Cache für eine bestimmte Applikation gespeicherter Daten handeln, andererseits um eine gezielte Abfrage der aktuellen Version eines bestimmten Objekts. Das Protokoll ist textbasiert. Im Fall der Aktualisierung aller Objekte werden folgende Nachrichten versandt:

1. ObjectCache → Update-Server
   Nachricht: `APP <AppID>`
   Die AppID ist die ID der Applikation, für die die Daten aktualisiert werden sollen.

2. ObjectCache ← Update-Server
   Nachricht: `APP <OK|NOK>`

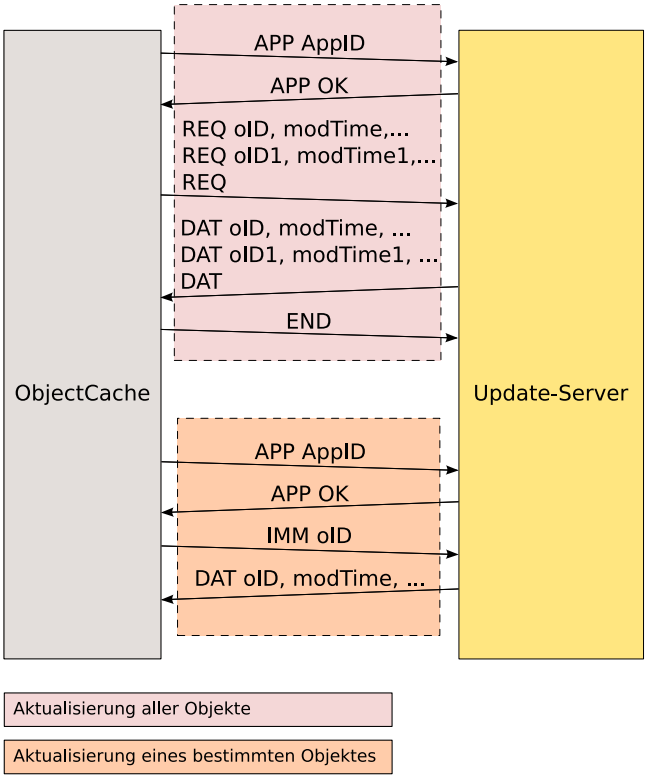

Abbildung 4: Protokoll zwischen Server und ObjectCache zum Abgleich der Daten

Der Update-Server überprüft, ob die durch die Anwendungs ID gegebene Anwendung durch den Update-Server unterstützt wird und teilt dies dem Client durch eine entsprechende Nachricht mit.

3. ObjectCache → Update-Server
   Nachricht: `REQ <objectID>, <modificationTime>, <objectType>` oder `REQ`
   Es wird jeweils mit einem Kommando eine Anfrage an den Cache nach einem bestimmten Objekt, an den Update-Server übermittelt. Die Übertragung der Liste der gestellten Anfragen, erfolgt durch Wiederholung des Kommandos. Ein leerer `REQ` Befehl ohne Parameter signalisiert das Ende der Liste. Alternativ kann auch ein spezieller Befehl, beispielsweise `REQEND` vorgesehen werden, der das Ende der Anfragen-Liste signalisiert.

4. ObjectCache ← Update-Server
   Nachricht: `DAT <objectID>`, `<modificationTime>`, `<objectType>`, `<objectData>` oder `DAT` bzw. `DATEND`
   Die Übertragung der Liste der aktualisierten Objekte wird durch die Wiederholung von `DAT` realisiert. Das Ende der Liste der zu aktualisierenden Objekte wird durch ein `DAT` Kommando ohne Parameter signalisiert. Alternativ kann auch ein spezieller Befehl, beispielsweise `DATEND` vorgesehen werden, der das Ende der Objekt-Liste signalisiert.

5. ObjectCache → Update-Server
   Nachricht: `END`
   Signalisiert dem Server, dass die Aktualisierung beendet ist und die Verbindung getrennt werden kann.

Für den Fall des gezielten Aktualisierens bzw. Abfragens eines Objekts, werden folgende Nachrichten ausgetauscht:

1. ObjectCache → Update-Server
   Nachricht: `APP <AppID>`
   Die AppID ist die ID der Applikation, für die die Daten aktualisiert werden sollen.

2. ObjectCache ← Update-Server
   Nachricht: `<APP|NOK>`
   Der Update-Server überprüft, ob die durch die Anwendungs ID gegebene Anwendung durch den Update-Server unterstützt wird und teilt dies dem Client durch eine entsprechende Nachricht mit.

3. ObjectCache → Update-Server
   Nachricht: `IMM <objectID>`
   Der ObjectCache übermittelt dem Update-Server die Objekt-ID des zu übertragenden Objektes.

4. ObjectCache ← Update-Server
   Nachricht: `DAT <objectID>`, `<modificationTime>`, `<objectType>`, `<objectData>`

Der Update-Server schickt die Daten des angefragten Objektes an den Client. Sollten keine Daten für das Objekt vorliegen, ist die Liste der Parameter leer.

7 Anwendung

Um den ObjectCache in Anwendungen verwenden zu können, muss man mittels der `ObjectCacheFactory` eine neue Instanz der ObjectCache-Klasse erzeugen. Dazu kann man, wie in Abschnitt 5.1 erwähnt, Vergleiche für die regulären Ausdrücke und die Übereinstimmungen mit den Objekt-IDs angeben. Über die Methoden der Anwendungsschnittstelle wird auf den Cache, wie in Abschnitt 4.2.2 beschrieben, zugegriffen.

Als eine Beispiel-Anwendung zur Nutzung des Caches, habe ich eine Shell programmiert, mit Hilfe derer man Objekte in den Cache laden, wieder auslesen oder löschen kann. Eine Übersicht über die in der Shell möglichen Befehle ist in Tabelle 4 gegeben. Wenn man ein ausführbares Programm in den Cache lädt, ermöglicht es die Shell den Programmcode direkt aus dem ObjectCache heraus auszuführen. Dazu ruft die Shell ein weiteres Programm auf, welches nicht im Cache liegen darf. Dieses Programm ist ein Anwendungslader und startet die eigentliche Anwendung. Das Starten des Anwendungsladers und der Ablauf des geladenen Programms findet in einem neuen Prozess statt. Es ist somit möglich parallel mehrere Anwendungen aus dem Cache zu starten. Die einzelnen Anwendungen nutzen wiederum die ObjectCache-Bibliothek, um ihrerseits Daten im Cache abzulegen oder aus dem Cache auszulesen.

7.1 Beispiel

Am Beispiel einer einfachen Anwendung möchte ich die Vorgehensweise beim Verwenden des ObjectCache zeigen. Abbildung 5 zeigt einen Screenshot dazu.

Zu Beginn muss eine SQLite-Datenbank erzeugt werden. Dazu wird der SQLite-Kommandozeilen-Client wie folgt aufgerufen:

```
$ sqlite3 -init create_database.sql objectCache.db
```

Die Datei `create_database.sql` muss im aktuellen Verzeichnis liegen und enthält SQL-Anweisungen die eine Datenbank, mit dem vom ObjectCache benötigten Schema, erzeugen. Die eigentlich Datenbank `objectCache.db` muss dann im gleichen Verzeichnis, wie die ObjectCache-Anwendung liegen.

Damit eine Shell beim Starten des ObjectCache ausgeführt werden kann, muss diese noch in die Datenbank eingefügt werden. Angenommen die Shell liegt als Programmdatei `ObjectCacheShell.exe` im aktuellen Pfad, lädt der Aufruf

```
$ mono AssemblyLoader.exe ObjectCacheShell.exe ObjectCache.exe
```

die Shell in die Datenbank. Der `AssemblyLoader` wird benötigt, um die Shell initial in die Datenbank zu laden. Die Datei `ObjectCache.exe` ist eine von der Shell benötigte Assembly – in diesem Fall die ObjectCache-Bibliothek selbst. Weitere benötigte Assemblies können als zusätzliche Parameter angegeben werden und werden dann später auch automatisch mit geladen.

Befehl und Argumente	Beschreibung
`close_cache`	Schließt die gerade geöffnete ObjectCache-Instanz.
`exit`	Schließt die Shell.
`get <Objekt-ID>`	Ruft das Objekt mit der gegebenen Objekt-ID aus dem Cache ab.
`ids_matching <Regulärer Ausdruck>`	Listet alle Objekt-IDs im gerade offenen Cache auf, die auf den gegebenen regulären Ausdruck passen.
`list_apps`	Listet alle Anwendungen auf, die im gerade offenen Cache gespeichert sind. Anwendungen sind Objekte, deren Objekt-IDs mit dem Präfix `app.` versehen sind.
`load <Objekt-ID>`	Führt ein Programm aus, welches unter der gegebenen Objekt-ID im gerade offenen Cache gespeichert ist. Anwendungen besitzen den Typ `System.Byte[]`.
`new_cache <Anwendungs-ID>`	Öffnet eine neue Instanz des ObjectCache mit der gegebenen Anwendungs-ID.
`put <Objekt-ID> <Objekt> <Objekt-Typ>`	Fügt das gegebene Objekt, unter der gegebenen Objekt-ID, mit dem gegebenen Objekt-Typ in den gerade geöffneten ObjectCache ein. Auf der Kommandozeile können nur Zeichenketten angegeben werden, weshalb sich mit diesem Kommando effektiv nur Strings in den Cache einfügen lassen.
`put_file <Objekt-ID> <Dateipfad>`	Fügt die angegebene Datei, unter der angegebenen Objekt-ID in den gerade offenen ObjectCache ein.
`rm|del|remove|delete <Objekt-ID>`	Löscht das Objekt mit der gegebenen Objekt-ID aus dem gerade geöffneten Cache.
`status`	Zeigt den aktuellen Status an, d.h., ob gerade eine Cache-Instanz offen ist und wenn ja, mit welcher Anwendungs-ID diese Instanz geöffnet wurde.

Tabelle 4: Übersicht über die Befehle der ObjectCacheShell

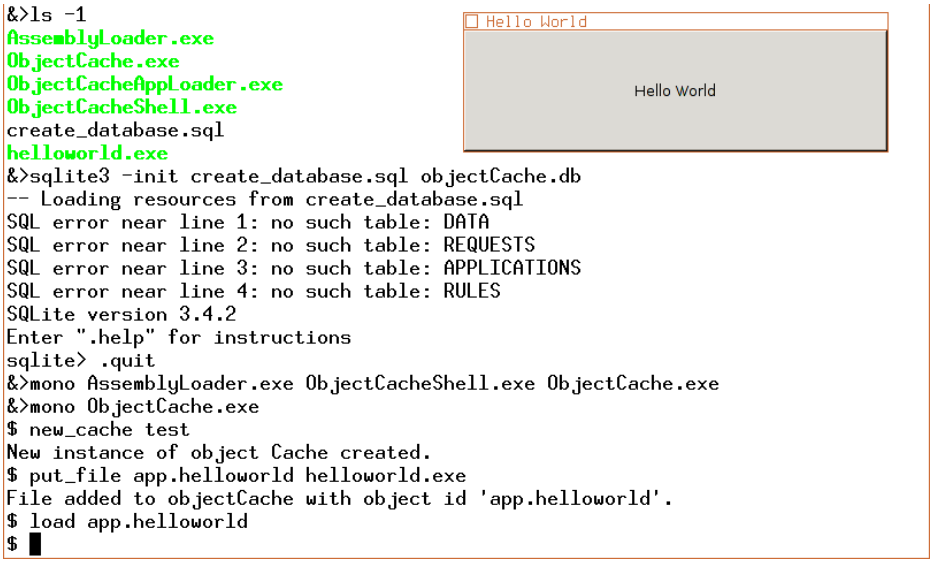

Abbildung 5: Beispiel für die Benutzung der ObjectCacheShell

Ist die Shell in den Cache eingefügt, kann der ObjectCache mittels des Kommandos

```
$ mono ObjectCache.exe
```

geladen werden. Mit dem Aufruf wird die Shell geladen. In dieser können dann weitere Programme eingefügt werden.

Wenn man jetzt z.B. eine GTK-HelloWorld-Anwendung in den Cache laden will, kann dies in der ObjectCacheShell über das Kommando `put_file` erfolgen. Nehmen wir an, die Anwendung heißt `helloworld.exe` und soll unter der ObjectID `app.helloworld` gespeichert werden. Um Objekte in den Cache laden kann, muss dieser zuerst mit einer Anwendungs-ID geöffnet werden. Diese sei `test`. Das Öffnen geschieht durch

```
$ new_cache test
```

Anschließend kann die Anwendung mittels des Kommandos

```
$ put_file app.helloworld helloworld.exe
```

in den Cache eingefügt werden. Soll die Anwendung ausgeführt werden, genügt es in einem geöffnetem Cache mit der entsprechenden Anwendungs-ID

```
$ load app.helloworld
```

aufzurufen und die Anwendung wird in einem neuen Prozess gestartet.

Allgemein, muss vor der ersten Aktion in der Shell, ein Cache mittels `new_cache` geöffnet werden. Danach können mit den entsprechenden Befehlen Objekte hinzugefügt, ausgelesen und entfernt werden. Wie oben beschrieben können dann auch Anwendungen aus dem Cache heraus gestartet werden.

8 Mögliche Erweiterungen

Der nun folgende Abschnitt gibt einen Überblick über mögliche Erweiterungen der in dieser Studienarbeit beschriebenen Version des ObjectCache.

Ein für den Nutzer des ObjectCache relativ aufwendige Anforderung ist die der Implementation eines Vergleichs für Übereinstimmungen und reguläre Ausdrücke. Um trotzdem eine Rangfolge festlegen zu können, wäre es denkbar den einzelnen Regel vom Nutzer fest eine numerische Priorität zuordnen zu lassen. Dies verhindert zwar, dass nach Inhalt der Übereinstimmungen verglichen werden kann, vereinfacht aber die Benutzung des ObjektCache sehr.

Eine weitere Möglichkeit der Erweiterung besteht in der Komponente, die den Abgleich der Daten vornimmt. Das bis jetzt recht simple Protokoll könnte durch eine eventuell standardisierte Lösung, wie z.B SyncML[2] ersetzt werden. Wie schon angesprochen, könnte es später möglich sein, beim Update-Server gezielt neue Dienste zu registrieren, für die dieser dann Daten zwischenspeichert. Dazu wäre es notwendig die Art der Schnittstelle des zwischenzuspeichernden Dienstes zu übermitteln. So z.B. ob die Daten über einen Web-Service abgegriffen werden sollen oder ob es notwendig ist direkt die HTTP-Seiten zu parsen.

Auch der Umgang mit den Anwendungs-IDs kann noch geändert werden, so dass es einer Anwendung nicht möglich ist die Daten einer anderen Anwendung zu lesen oder zu manipulieren. Dies könnte zum Beispiel über eine Art Passwort erfolgen, dass die Anwendung bei der Registrierung festlegt und das bei der Anforderung einer ObjectCache-Instanz angegeben werden muss. Möglich wäre eine Variante, in der die Anwendung bei der Registrierung einen öffentlichen Schlüssel hinterlegt. Bei der Anfrage nach einer ObjectCache-Instanz wird dann ein Challenge-Token von der Anwendung signiert. Über diese Signatur wird dann die Anwendung authentifiziert.

Insgesamt ist bis jetzt auch die Frage der Sicherheit noch nicht in der Architektur berücksichtigt worden. Weiterführend ist es aber von großer Bedeutung, dass das System ein Mindestmaß an Sicherheit garantiert, damit es von den Benutzer angenommen wird. In diesen Themenbereich fallen unter anderem die Übermittlung von Login-Daten für Dienste und die Vergabe von Berechtigungen darüber, welche Anwendungen auf die mobile Gerät geladen werden können und welche Zugriffsberechtigungen diese erhalten.

Ein Problem das bis jetzt auch noch nicht im ObjectCache gelöst ist, betrifft eine Begrenzung der Größe des Caches. Auf einem mobilen Gerät ist Speicherplatz potentiell knapp und daher ist es sinnvoll den maximalen Speicherplatz für den Cache zu begrenzen. Mit dieser Einschränkung ergibt sich aber das Problem, dass Objekte eventuell aus dem Cache entfernt werden müssen. In [7] wird dabei ein Prefetching Cache Manager vorgestellt. Dieser berücksichtigt nicht nur wie bei der üblichen Least-Recently-Used-Strategie die Zugriffe in der Vergangenheit, sondern zieht auch die Wahrscheinlichkeit eines Zugriffs in der Zukunft in die Auswahl eines Objektes mit ein, dass aus dem Cache entfernt werden soll. Ein Vergleich einiger Strategien zur Entfernung von WWW-Seiten aus einem Cache findet sich in [11]. Im ObjectCache würde die „Wartung“ des Caches in einem separaten Thread implementiert werden. In der bisherigen Implementation wird zwar die Lebensdauer eines Objektes berücksichtigt, dieses aber nicht nach Ablauf der

Gültigkeit aus dem Cache entfernt.

9 Ähnliche Anwendungen - Google Gears

Eine dem ObjectCache ähnliche Anwendung, ist Teil von Google Gears[6]. Dies ist die Klasse `ResourceStore` aus dem Teil des `LocalServer` und ist dazu bestimmt, Elemente einer Web-Applikation lokal zwischenzuspeichern. Die Benutzung von Google Gears erfolgt über eine JavaScript-Schnittstelle des Browsers. Google Gears an sich ist in C++ implementiert. Auch der `ResourceStore` speichert die Objekt innerhalb einer SQLite-Datenbank. Die einzelnen Objekte werden dabei durch eine URL referenziert. Der Zugriff auf die einzelnen Objekt erfolgt bei Google Gears aber anders als im ObjectCache. Es gibt eine Instanz der Klasse `LocalServer`, die Objekte z.B. in einem `ResourceStore` speichert. Dieser `LocalServer` fängt Anfragen an URLs, für die Objekte im `ResourceStore` liegen, ab und beantwortet die Anfragen aus dem Cache. Dadurch erfolgen Anfragen nach Objekten absolut transparent. In Google Gears existieren jedoch keine Möglichkeiten auf einige Parameter des Caches einzugehen, wie dies beim ObjectCache möglich ist und über die Regeln realisiert wird. Diese Unterschiede ergeben sich aber direkt aus den verschiedenen Ausrichtungen von Google Gears und dem ObjectCache. Google Gears ist für die Zwischenspeicherung von Daten aus Web-Applikationen konzipiert. Der ObjectCache dagegen soll Objekte innerhalb von Mono-Programmen verwalten, was ein breites Spektrum von Anwendungsfällen umfasst.

10 Zusammenfassung

Der ObjectCache bietet in seiner jetzigen Form die grundlegenden Funktionen, die für das Caching von Mono-Objekte nötig sind. Darüber hinaus bietet er den Anwendungen durch die Regeln die Möglichkeit, gezielt auf das Verhalten des Caches Einfluss zu nehmen. Es existiert außerdem ein rudimentäres Protokoll, um den Inhalt des Caches mit einem Update-Server abzugleichen und zu aktualisieren. Um jedoch Precaching komplett umzusetzen ist es notwendig im Update-Server den Mechanismus des Prefetching zu implementieren. Als Anhaltspunkte können Veröffentlichungen zum Thema Prefetching und Caching im Bereich des WWW, zum Beispiel [4, 5, 6, 10], dienen. Erst durch das Zusammenspiel beider Komponenten entsteht eine Architektur, die dem Nutzer das Gefühl bietet, auch unterwegs – ohne Netzwerkanbindung – stets auf seine im Netz vorhandenen Daten zugreifen zu können.

Literatur

[1] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[2] Open Mobile Alliance. Syncml specifications, version 1.1.

[6] http://gears.google.com

[3] Michael Angermann. Analysis of speculative prefetching. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(2):13–17, 2002.

[4] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of the 4th ACM International Conference on Information and Knowledge Management*, Baltimore, MD, 1995.

[5] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[6] Brian D. Davison. A web caching primer. *IEEE Internet Computing*, 5(4):38–45, 2001.

[7] Jim Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer*, pages 197–207, 1994.

[8] Z. Jiang and L. Kleinrock. An adaptive network prefetch scheme, 1998.

[9] A.R. Meyer and L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. *13th Annual IEEE Symp. on Switching and Automata Theory*, 1972.

[10] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, pages 224–244. IEEE Computer Society Press and Wiley, New York, NY, 2001.

[11] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for World-Wide Web documents. In *Procedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

Reports published by Humboldt University Berlin, Computer Science Department, Systems Architecture Group.

1. SAR-PR-2005-01: Linux-Hardwaretreiber für die HHI CineCard-Familie. Robert Sperling. 37 Seiten.
2. SAR-PR-2005-02, NLE-PR-2005-59: State-of-the-Art in Self-Organizing Platforms and Corresponding Security Considerations. Jens-Peter Redlich, Wolf Müller. 10 pages.
3. SAR-PR-2005-03: Hacking the Netgear wgt634u. Jens-Peter Redlich, Anatolij Zubow, Wolf Müller, Mathias Jeschke, Jens Müller. 16 pages.
4. SAR-PR-2005-04: Sicherheit in selbstorganisierenden drahtlosen Netzen. Ein Überblick über typische Fragestellungen und Lösungsansätze. Torsten Dänicke. 48 Seiten.
5. SAR-PR-2005-05: Multi Channel Opportunistic Routing in Multi-Hop Wireless Networks using a Single Transceiver. Jens-Peter Redlich, Anatolij Zubow, Jens Müller. 13 pages.
6. SAR-PR-2005-06, NLE-PR-2005-81: Access Control for off-line Beamer – An Example for Secure PAN and FMC. Jens-Peter Redlich, Wolf Müller. 18 pages.
7. SAR-PR-2005-07: Software Distribution Platform for Ad-Hoc Wireless Mesh Networks. Jens-Peter Redlich, Bernhard Wiedemann. 10 pages.
8. SAR-PR-2005-08, NLE-PR-2005-106: Access Control for off-line Beamer Demo Description. Jens Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge. 18 pages.
9. SAR-PR-2006-01: Development of a Software Distribution Platform for the Berlin Roof Net (Diplomarbeit / Masters Thesis). Bernhard Wiedemann. 73 pages.
10. SAR-PR-2006-02: Multi-Channel Link-level Measurements in 802.11 Mesh Networks. Mathias Kurth, Anatolij Zubow, Jens Peter Redlich. 15 pages.
11. SAR-PR-2006-03, NLE-PR-2006-22: Architecture Proposal for Anonymous Reputation Management for File Sharing (ARM4FS). Jens Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge, Torsten Dänicke. 20 pages.
12. SAR-PR-2006-04: Self-Replication in J2me Midlets. Henryk Plötz, Martin Stigge, Wolf Müller, Jens-Peter Redlich. 13 pages.
13. SAR-PR-2006-05: Reversing CRC – Theory and Practice. Martin Stigge, Henryk Plötz, Wolf Müller, Jens-Peter Redlich. 24 pages.
14. SAR-PR-2006-06: Heat Waves, Urban Climate and Human Health. W. Endlicher, G. Jendritzky, J. Fischer, J.-P. Redlich. In: Kraas, F., Th. Krafft & Wang Wuyi (Eds.): Global Change, Urbanisation and Health. Beijing, Chinese Meteorological Press.
15. SAR-PR-2006-07: 无线传感器网络研究新进展 (State of the Art in Wireless Sensor Networks). 李刚 (Li Gang), 伊恩斯•彼得•瑞德里希 (Jens Peter Redlich)

16. SAR-PR-2006-08, NLE-PR-2006-58: Detailed Design: Anonymous Reputation Management for File Sharing (ARM4FS). Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Martin Stigge, Christian Carstensen, Torsten Dänicke. 16 pages.
17. SAR-PR-2006-09, NLE-SR-2006-66: Mobile Social Networking Services Market Trends and Technologies. Anett Schülke, Miquel Martin, Jens-Peter Redlich, Wolf Müller. 37 pages.
18. SAR-PR-2006-10: Self-Organization in Community Mesh Networks: The Berlin RoofNet. Robert Sombrutzki, Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 11 pages.
19. SAR-PR-2006-11: Multi-Channel Opportunistic Routing in Multi-Hop Wireless Networks. Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 20 pages.
20. SAR-PR-2006-12, NLE-PR-2006-95: Demonstration: Anonymous Reputation Management for File Sharing (ARM4FS). Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen, Torsten Dänicke. 23 pages.
21. SAR-PR-2006-13, NLE-PR-2006-140: Building Blocks for Mobile Social Networks Services. Jens-Peter Redlich, Wolf Müller. 25 pages.
22. SAR-PR-2006-14: Interrupt-Behandlungskonzepte für die HHI CineCard-Familie. Robert Sperling. 83 Seiten.
23. SAR-PR-2007-01: Multi-Channel Opportunistic Routing. Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 10 pages. IEEE European Wireless Conference, Paris, April 2007.
24. SAR-PR-2007-02: ARM4FS: Anonymous Reputation Management for File Sharing. Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen, 10 15 pages.
25. SAR-PR-2007-03: DistSim: Eine verteilte Umgebung zur Durchführung von parametrisierten Simulationen. Ulf Hermann. 26 Seiten.
26. SAR-SR-2007-04: Architecture for applying ARM in optimized pre-caching for Recommendation Services. Jens-Peter Redlich, Wolf Müller, Henryk Plötz, Christian Carstensen. 29 pages.
27. SAR-PR-2007-05: Auswahl von Internet-Gateways und VLANs im Berlin RoofNet. Jens Müller. 35 Seiten.
28. SAR-PR-2007-06: Softwareentwicklung für drahtlose Maschennetzwerke – Fallbeispiel: BerlinRoofNet. Mathias Jeschke, 48 Seiten.
29. SAR-SR-2007-07, NLE-SR-2007-88: Project Report: Anonymous Attestation of Unique Service Subscription (AAUSS). Jens-Peter Redlich, Wolf Müller, 19 pages.
30. SAR-PR-2007-08: An Opportunistic Cross-Layer Protocol for Multi-Channel Wireless Networks. Anatolij Zubow, Mathias Kurth, Jens-Peter Redlich, 5 pages. 18th IEEE PIMRC, Athens, Greece, 2007.

31. SAR-PR-2007-09: 100% Certified Organic: Design and Implementation of Self-Sustaining Cellular Networks. Nathanael A. Thompson, Petros Zerfos, Robert Sombrutzki, Jens-Peter Redlich, Haiyun Luo. ACM HotMobile'08. Napa Valley (CA), United States, Feb 25-26, 2008.
32. SAR-SR-2007-10, NLE-SR-2007-88: Project Report: Summary of encountered Security / Performance / Scalability problems with uPB and Wireless Thin Client Architecture. Jens-Peter Redlich, Wolf Müller, 26 pages.
33. SAR-PR-2008-01:On the Challenges for the Maximization of Radio Resources Usage in WiMAX Networks. Xavier Perez-Costa*, Paolo Favaro*, Anatolij Zubow, Daniel Camps* and Julio Arauz*, Invited paper to appear on 2nd IEEE Broadband Wireless Access Workshop colocated with IEEE CCNC 2008. *NEC Laboratories Europe, Network Research Division, Heidelberg, Germany
34. SAR-PR-2008-02: Cooperative Opportunistic Routing using Transmit Diversity in Wireless Mesh Networks. Mathias Kurth, Anatolij Zubow, Jens-Peter Redlich. 27th IEEE INFOCOM, Phoenix, AZ, USA, 2008.
35. SAR-PR-2008-03: Evaluation von Caching-Strategien beim Einsatz von DHTs in drahtlosen Multi-Hop Relay-Netzen - Am Beispiel eines verteilten Dateisystems. Felix Bechstein. Studienarbeit.