Diplomarbeit

Design und Evaluation eines Offline-Kartendienstes auf der Grundlage von Openstreetmap unter Verwendung von Precachingverfahren

Christian Otto

13. Dezember 2009

Institut für Informatik
Lehrstuhl für Systemarchitektur

Betreuer: Prof. Dr. Redlich

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Berlin, den 13. Dezember 2009

Unterschrift

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Institutes für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 13. Dezember 2009

Unterschrift

Abstrakt

Mobile Computer wie z.B. Netbooks, PDAs und Smartphones haben in den letzten Jahren sehr an Bedeutung gewonnen. Sie bieten dem Anwender die Möglichkeit, potentiell zu jeder Zeit und von jedem Ort aus das Internet zu erreichen. Bei genauer Betrachtung sind dabei viele Probleme ungelöst. Einerseits sind schnelle Internetverbindungen von mobilen Geräten, z.B. via UMTS, für den Nutzer mit einigen Kosten verbunden, andererseits steigt die mobile Netzwerklast mit zunehmender mobiler Nutzung des Internets dramatisch an. Aus beiden Gründen sind mehr und mehr Technologien gefragt, die die Netzwerklast reduzieren, jedoch trotzdem die Funktionalität einer mobil genutzten Internetanwendung aufrechterhalten. Häufig genutzte Webanwendungen sind z.B. geografische Kartendienste wie Google Maps, Yahoo Maps, Virtual Earth oder auch Openstreetmap. Diese klassischen Webanwendungen erzeugen durch das Übertragen von Karten als Bilddateien eine hohe Netzwerklast, was bei mobiler Nutzung zu den erwähnten Problemen führt. Ein lohnender Beitrag zur Lösung wäre die Beschränkung der vom Server zum Client übermittelten Daten auf das zur Funktionalität unbedingt Notwendige. Somit ist es sinnvoll, Daten möglichst genau auszuwählen, bevor sie zum Client übertragen werden.

Im Sinne dieser Strategie wird hier ein mobiler, geografischer Kartendienst zur Ausführung auf mobilen Geräten als Offline-Webanwendung konzipiert. Grundlage dafür bildet das offene Projekt *Openstreetmap*. Dabei stehen Fragen zu intelligenten Cachingverfahren zur Versorgung von Clients mit zukünftig benötigten Daten im Vordergrund. Es werden verschiedene Ansätze für das vorausschauende Caching von geografischen Kartendaten vorgestellt und erläutert, die im Design einer hier entwickelten experiementellen Architektur zur Anwendung kommen und abschließend evaluiert werden.

Inhaltsverzeichnis

1 Einführung

Zum besseren Verständnis der im Abschnitt 1.3 dargestellten Problemstellung der vorliegenden Arbeit wird zunächst auf einige grundlegende Aspekte der standortbezogenen Internet-Nutzung eingegangen.

1.1 Mobile Internetnutzung

In den letzten Jahren hat der Marktanteil leistungsfähiger mobiler Rechner stark zugenommen. Das Sortiment mobiler Rechner erstreckt sich von Notebook und Netbook über Personal Digital Assistant (PDA) bis hin zum Smartphone. Die Leistungsfähigkeit von Geräten der beiden letztgenannten Kategorien ist in den letzten Jahren stark gestiegen, wodurch diese in der Lage sind, immer anspruchsvollere Anwendungen auszuführen. Oft verfügen diese Geräte über Wireless LAN-, UMTS- oder GSM-Schnittstellen[1], mit denen eine Verbindung zum Internet möglich ist. Somit sind Webservices, die früher hauptsächlich oder ausschließlich über stationäre Rechner zu erreichen waren, dem Nutzer von potentiell jedem beliebigen Ort auf der Welt zugänglich.

Eines der Hauptprobleme bei der Nutzung von internetbasierten Diensten auf mobilen Geräten ist die unvorhersagbare und unzuverlässige Konnektivität zum Internet sowie entstehende Kosten pro Datenvolumen. Für die Nutzbarkeit bzw. Nutzerakzeptanz dieser Dienste ist der Grad der Transparenz in Bezug auf die Änderungen der Art (Handover zwischen verschiedenen Netzwerken) bzw. des Status der Internetverbindung ein wichtiges Kriterium. Um diese Transparenz zu gewährleisten, ist eine Lösung notwendig, die lokal Daten für den zeitweiligen Offlinebetrieb des jeweiligen Dienstes bereitstellt. Auf diese Weise können lesende Operationen auf der Basis lokal vorliegender Daten ausgeführt und schreibende Operationen vorgemerkt werden, um die Übertragung zu einem geeigneten Zeitpunkt bei vorhandener Internetverbindung zum Webservice ausführen zu können.

Wichtige beschränkende Faktoren bei der Erstellung mobiler Anwendungen liegen in der Prozessor- und Speicherkapazität mobiler Geräte sowie der Kapazität von Akkumulatoren. Die Laufzeit der Energiespeicher nimmt mit zunehmender Rechenbelastung eines mobilen Geräts ab. Daher muss eine Balance zwischen der Rechenbelastung des Clients und seiner Funktionalität gefunden werden.

Weiterhin kommen durch die Mobilität der Clients von Webservices nun auch Anwendungsfälle hinzu, bei denen die aktuelle geografische Position des Anwenders ein wichtiger Parameter

[1] Local Area Network (LAN), Universal Mobile Telecommunications System (UMTS), Global System for Mobile Communications (GSM)

einer Anfrage an den Webservice darstellt. Durch die Abhängigkeit des Ergebnisses der Anfrage von der Position des Clients verändert sich die Anwendungslogik maßgeblich. Derartige Dienste werden als standortbezogene Dienste oder Location Based Service (LBS) bezeichnet.

Gängige geografische Kartendienste sind z.B. Google Maps, Yahoo Maps, Virtual Earth und Openstreetmap. Diese stellen klassische Webservices im Internet dar und liefern Funktionen wie:

- Darstellung von Karten in verschiedenen Zoomstufen: Zoomstufen sind ganzzahlige Maßstabsskalierungen für ein dargestelltes geografisches Gebiet. Bei Openstreetmap bedeutet die Erhöhung der Zoomstufe um eine Einheit die Halbierung des Maßstabs in jeder Richtung, also die Vervierfachung der Anzahl der für die Darstellung des gleichen Gebiets notwendigen *Kacheln*, wobei mit Kachel jetzt und in Zukunft eine Bilddatei aus dem Fundus des Kartendienstes mit stets konstanter Kantenlänge (Pixelanzahl) bezeichnet wird. Mit ansteigenden Zoomstufen steigt entsprechend die Detailgenauigkeit der Darstellung.
- Suche nach geografischen Objekten (z.B. Straßen, Kreuzungen, Gebäuden, Points of Interest[2])
- Routenplanung bei bei Eingabe von Start- und Zieladresse
- Möglichkeit zur Verlinkung von Punkten auf der Karte zu diversen Seiten im Internet (z.B. Wikipedia, Webcams u.a.)

Um eine dieser Funktionen zu nutzen, ist bisher eine bestehende Verbindung zum Internet, d.h. zum Serviceprovider notwendig. Eine Anfrage nach Kartenmaterial an Google Maps für ein begrenztes geografisches Gebiet wird durch das Senden von Kartenkacheln und Metadaten durch den Google Maps Server beantwortet. Um dabei das Antwortverhalten der Weboberfläche der Anwendung im Webbrowser dynamisch zu halten, werden für derartige zeitintensive Abfragen asynchrone Anfragen (Asynchronous JavaScript and XML (AJAX)) verwendet. Ladevorgänge sind somit für den Nutzer weitestgehend transparent gehalten, sodass der Eindruck erweckt werden könnte, die Anwendung liefe vollständig lokal, jedoch zieht ein Zusammenbruch der Internetverbindung zwischen Client und Google Maps-Server auch einen Ausfall des Dienstes auf dem Client nach sich.

1.2 Das Openstreetmap-Projekt

Openstreetmap (OSM)[3] ist ein Wiki-Projekt zur Erstellung einer weltweiten, freien und möglichst vollständigen geografischen Karte. Für einige geografische Bereiche wurden die Daten durch Importe aus anderen Projekten in die OSM-Datenbank aufgenommen. So stammt ein Teil der Openstreetmap zugrundeliegenden Daten aus Datenimporten folgender Quellen:

- TIGER-Datenbank als Basis für Karten in den USA
- Datenbanken der Firma AND für niederländische, chinesische und indische Karten

[2] *Orte von Interesse* sind punktförmige Markierungen auf der Karte, die ein für den Nutzer möglicherweise bedeutungsvollen Ort oder ein Objekt ausweisen.
[3] URL: www.openstreetmap.org

- National Geospatial-Intelligence Agency (NGA) für Küstenverläufe
- CIA World DataBank III für Ländergrenzen
- weitere Datenimporte, siehe http://wiki.openstreetmap.org/wiki/Potential_Datasources

Wie bei jedem Wiki-Projekt wird ansonsten der Inhalt, der hier im Gegensatz zu z.B. Wikipedia aus geografischen Daten besteht, durch das Wirken der Mitglieder des Projekts erstellt. Zu diesen Daten gehören u.a. Beschreibungen von Wegetrajektorien (im Folgenden *Traces* genannt), die von Nutzern zum OSM-Server hochgeladen werden. Anhand dieser Traces können mit Hilfe von Programmen wie JOSM, Merkaator oder Potlatch[4] Wege oder Umrisse von Objekten nachgezeichnet werden, um so die Karte zu komplettieren. Änderungen können auch ohne die Nutzung von Traces "aus dem Gedächtnis" anhand von Orientierungspunkten (z.B. Kirchtürme etc.) erfolgen. Trace-Daten bilden im Weiteren eine Grundlage für den Evaluationsteil (Kapitel 6) der vorliegenden Arbeit. Die Daten der Openstreetmap-Datenbank dürfen nach der Creative Commons Attribution-ShareAlike 2.0-Lizenz[5] verwendet werden.

1.3 Problemstellung der Arbeit

In dieser Arbeit wird eine Client-Server-Architektur für einen Kartendienst, basierend auf Openstreetmap vorgestellt, die auch im Fall einer zeitlich beschränkten und unvorhersehbar wiederkehrenden Internetverbindung des Clients eine Verfügbarkeit des Dienstes sicherstellen soll. Deshalb ist es das primäre Anliegen, Möglichkeiten zur optimalen Nutzung einer vorhandenen Internetverbindung zu untersuchen. Eine erste Möglichkeit dazu besteht in der Komprimierung der zu übertragenden Daten zur Verringerung sowohl der Netzwerklast als auch der Übertragungszeit. Um die potentiell teure und nur sporadisch vorhandene Internetverbindung darüber hinaus bestmöglich zu nutzen, sollen hier Methoden untersucht werden, mit denen durch geschickte Selektion der zu übertragenden Daten deren Volumen weiter eingeschränkt bzw. mit denen bei gleichem Datenvolumen die Qualität, d.h. der Nutzwert der Daten, erhöht werden kann. Bei einem Zusammenbruch der Internetverbindung eines mobilen Geräts kann der Dienst ausschließlich mit den Daten weiterarbeiten, die zuvor heruntergeladen wurden, d.h. replizierte Daten eines entfernten Servers müssen in einem lokalen Cache verfügbar gehalten werden.

Im ursprünglichen Sinne sind Caches schnelle Speicher, die die Latenzzeit beim Zugriff auf Daten in einem langsameren Speicher verringern sollen. Diese schnellen Speicher sind meist nicht nur wesentlich schneller, sondern auch wesentlich kleiner (und teurer) als der langsame Speicher, für den sie als Puffer dienen. Sobald auf Daten des langsamen Speichers zugegriffen wird, werden diese zunächst in den Cache geladen und sind somit bei Zugriffen in (naher) Zukunft schneller verfügbar, da sie nicht mehr aus dem langsamen Speicher abgerufen werden

[4]Die Karteneditoren JOSM und Merkaator unterscheiden sich sowohl im Nutzerinterface als auch in der Funktionalität. Während Merkaator von beiden Editoren selbsterklärender bedienbar ist, kann JOSM eine wesentlich umfangreichere Funktionalität aufweisen. Potlatch ist der unter www.openstreetmap.org verfügbare auf Flash basierende Editor.

[5]Die Creative Commons Attribution-ShareAlike 2.0-Lizenz erlaubt das Ändern, Kopieren und Verteilen eines Produkts, das unter dieser Lizenz veröffentlicht wurde, bei Nennung des Autors auf die von ihm vorgegebene Art und Weise. Geänderter lizensierter Inhalt muss unter den gleichen oder unter vergleichbaren Bedingungen weitergegeben werden, es sei denn, der Rechteinhaber stimmt ihrer Aufhebung zu. ([cre])

müssen. Benutzt wird hier das *Prinzip der zeitlichen Lokalität*, nach dem Daten, die in der nahen Vergangenheit benutzt wurden, wahrscheinlich auch in der nahen Zukunft benötigt werden. Oft werden, wie z.B. beim Festplattencaching, mehr Daten in den Cache geschrieben als angefragt wurden. Im Falle des Festplattencachings werden so nicht nur die Daten des angefragten Sektors in den Festplattencache geschrieben, sondern auch die der folgenden oder vorhergehenden Sektoren (*read ahead* oder *read behind*). Nach dem *Prinzip der örtlichen Lokalität* werden diese Daten in naher Zukunft wahrscheinlich benötigt.

Heutzutage sind Caches auch im Internet anzufinden. Web Caches reduzieren die Netzwerklast und Latenz beim Herunterladen von Webinhalten, indem sie Webinhalte lokal speichern und wiederholte Anfragen nach gleichen Webinhalten lokal bedienen können. Sowohl moderne Webbrowser als auch Proxy-Server, wie z.B. *Squid*, unterstützen diese Technologie.

Im Gegensatz zu Speichercaches (z.B. Festplattencache) greift beim Webcache das Prinzip der örtlichen Lokalität nicht ohne Weiteres. Während beim Festplattencaching aufeinander folgende Sektoren in vielen Fällen einen Bezug zueinander haben (z.B. zur gleichen Datei gehören), sind Bezüge zwischen Web-Dokumenten nur schwer herzustellen. Einzelne Webseiten, wie z.B. die einiger Online-Shops, betreiben deshalb inhalts- und/oder bewertungsbasierte Empfehlungssysteme. Dies geschieht, um Produkte zu empfehlen, die dem zuletzt betrachteten ähneln oder für das sich andere Kunden neben dem aktuell betrachteten Produkt ebenfalls interessiert haben.

Zur Entwicklung einer Offline-Applikation, deren Grunddatenbestand aus Web-Inhalten besteht, ist jedoch in vielen Fällen die Fähigkeit eines "read ahead" notwendig. Im Kontext einer Webanwendung bedeutet dies ein Lesen von Webinhalten im Voraus, die mit einer gewissen Wahrscheinlichkeit in naher Zukunft benötigt werden. Dieser Vorgang wird im Weiteren als *Precaching* bezeichnet. Der Offlinebetrieb des Web-Kartendienstes *Openstreetmap* erfordert das lokale Speichern von geografischen Daten bzw. Kartenmaterial in Form von Bilddateien (Kacheln). Diese Daten sind über die geografischen Position bzw. Region, die sie abbilden, referenzierbar. Diese Positionen sind zweidimensionale Koordinaten und markieren Längen- und Breitengrad eines definierten Punktes (z.B. eines Eckpunktes) der Kachel. Ausgehend davon, dass ein Client, der die Offline-Applikation nutzt, sich nicht sprunghaft, sondern kontinuierlich über das Koordinatensystem bewegt, ist hier die Anwendung des Prinzips der örtlichen Lokalität plausibel, allerdings nicht in einer Dimension (*read ahead* oder *read behind*), sondern zweidimensional (Längengrad, Breitengrad). Ein sich kontinuierlich bewegender Client wird stets Daten benötigen, die sich im Längen-/Breitengrad-System auf die "in der Nähe" liegende Umgebung seiner eigenen Position beziehen und somit "in der Nähe" der Daten liegen, die er früher schon angefragt hat. Dies lässt vermuten, dass ein Precaching hier möglich ist, was in dieser Arbeit auf der Grundlage verschiedener Annahmen überprüft wird. Dazu wird mit Hilfe von Anfrageparametern wie der aktuellen Position eines Clients und seines Geschwindigkeitsvektors sowie zusätzlichen statistischen Informationen die Serverseite des Dienstes in die Lage versetzt, abschätzen zu können, welche Kartenkacheln ein Client in der nächsten Zukunft wahrscheinlich benötigen wird. Die diesen Kartensegmenten zugeordneten Informationen werden zum mobilen Gerät übermittelt und für die spätere Benutzung gespeichert (Precaching). Da heutzutage viele mobile Geräte über einen integrierten GPS-Empfänger verfügen, kann die

Kenntnis der aktuellen Position, Geschwindigkeit und Bewegungsrichtung seitens des mobilen Gerätes vorausgesetzt werden.

Das Hauptproblem besteht somit in der Entwicklung von intelligenten Precachingverfahren, die aufgrund objektiver Kriterien die subjektive Entscheidung des Clients über seinen realen Weg bestmöglich vorwegnehmen. Die Trefferquote für "wahre" Vorhersagen kann nur statistisch beurteilt werden. Der Schwerpunkt dieser Arbeit liegt in der Betrachtung verschiedener Precaching-Verfahren und ihrer Evaluierung. Es werden zwei grundsätzlich verschiedene Verfahrenstypen untersucht (siehe 3.4.3):

- geometrische Precachingverfahren
  - kreisförmiges Precaching
  - elliptisches Precaching
  - kreissektorförmiges Precaching
- nicht-geometrische Precachingverfahren
  - Precaching, basierend auf Potentialen
  - Precaching, basierend auf Ähnlichkeit von Umgebungen

Der Qualitätsparameter für ein Precachingverfahren wird von der Zählung von *Cache Hits* und *Cache Misses* abgeleitet. Bei dem Zugriff auf lokale Daten kommt es zu einem Cache Hit, falls sich der Client über ein Gebiet bewegt, von dem lokal Karteninformationen vorliegen. Bei einem Cache Miss liegen lokal keine Daten über dieses Gebiet vor. Ein bestimmtes Precachingverfahren P_1 ist einem anderen P_2 als qualitativ überlegen anzusehen, wenn bei sonst vergleichbaren Parametern das Verhältnis von Cache Hits zur Gesamtanzahl der lesenden Zugriffe[6]

$$\frac{\#\text{Cache Hits}}{\#\text{Cache Hits} + \#\text{Cache Misses}}$$

bei P_1 grösser ist als das bei P_2. Dieses Verhältnis wird im Folgenden *Cache Hit Rate* genannt.

Zu Evaluationszwecken wurden sowohl ein Mobile Openstreetmap (MOSM)-Server als auch ein MOSM-Simulator implementiert. Der Server ist konfigurierbar, um mit jedem der oben genannten Precachingverfahren zu funktionieren. Ebenso sind Parameter der Precachingverfahren konfigurierbar (siehe Kapitel 5). Der Simulator dient der Verhaltenssimulation virtueller Clients. Informationen über von Nutzern real zurückgelegte Wege werden von Openstreetmap öffentlich als *Trace-Files* zur Verfügung gestellt. Ihr eigentlicher Verwendungszweck besteht darin, den Verlauf von z.B. Wegen und Straßen oder Umrissen von Objekten zu markieren, um diese besser kartografieren zu können. Die Trace-Daten werden im Kontext der hier durchgeführten Simulation benutzt, um Wege, die durch virtuelle Clients zurückgelegt werden, zu beschreiben. Repräsentanten mit extremen Besonderheiten ("Ausreißer" mit z.B. unrealistisch langsamer/schneller Fortbewegung, extrem kurze/lange zurückgelegte Wegstrecken, ...), werden vor der Simulation ausgesondert und verworfen.

[6]Das Zeichen '#' steht hier und im Weitern für 'Anzahl'.

In Kapitel 2 werden zunächst grundlegende Konzepte wie Location Based Services und geografische Informationssysteme eingeführt sowie existierende Technologien für die Erstellung von Offline-Webservices vorgestellt.

Die Ableitung des Designs des mobilen Kartendienstes aus gestellten Anforderungen erfolgt in Kapitel 3. Hier werden u.a. die Precachingverfahren erläutert, auf die sich ein Großteil der späteren Evaluation bezieht.

Kapitel 4 stellt in diesem Projekt wichtige verwendete Komponenten wie PostGIS und die Openstreetmap API vor.

Erläuterungen zur Implementation eines Server-Prototyps und eines Simulators zur Simulation virtueller Clients sind in Kapitel 5 enthalten.

In Kapitel 6 wird die erarbeitete Lösung abschließend unter dem Aspekt der Precachingverfahren evaluiert und bewertet.

2 Grundlagen

Das im Abschnitt 1.3 erklärte Grundanliegen dieser Arbeit ordnet sich als Teilaspekt des in der Literatur unter dem etwas unscharfen Begriff *Mobile Computing* beschriebenen Problemkreis ein. Deshalb wird in Abschnitt 2.1 dazu eine kurze systematische Übersicht gegeben, bevor in Abschnitt 2.2 das Konzept von *Location Based Services* (LBS) vorgestellt wird, in dessen Kategorie Openstreetmap als mobile Offline-Webanwendung, wie sie in dieser Arbeit vorgestellt wird, einzuordnen ist. LBS und sogenannte *Geografische Informationssysteme* sind eng miteinander verknüpft. Als Teil eines LBS ermöglichen es solche Systeme, geografische Informationen in die Ausführung des Dienstes mit einfließen zu lassen. Ein gern benutzes Beispiel hierfür ist ein ortsabhängiges Branchen-Verzeichnis, das bei der Suche nach einem Restaurant nicht Adressen sämtlicher bekannter Restaurants zurückliefert, sondern den Standort des anfragenden Nutzers und eventuell weitere geografische Informationen mit berücksichtigt. Dies wird im Abschnitt 2.3 näher erläutert. Anschließend werden Technologien wie hier relevante Teile von HTML 5 und Gears vorgestellt, die moderne Webbrowser dazu befähigen, mobile Offline-Webapplikationen auszuführen. Unter Nutzung dieser Technologien ist es möglich, eine Offlineanwendung für den geografischen Kartendienst Openstreetmap in der für den Nutzer gewohnten Umgebung, seinem (mobilen) Browser, auszuführen.

2.1 Kurzer Abriss zu Mobile Computing

Der unscharfe Begriff *Mobile Computing* bezeichnet Funktionsmuster, die sich im Zuge der Entwicklung drahtloser Netzwerke und mobiler Computer herausgebildet haben. Diese können, wie in [ea99] beschrieben, in die Bereiche *Mobile-Aware Adaption*, *Extended Client-Server Model* und *Mobile Data Access* untergliedert werden:

- *Mobile-Aware Adaption*: Dieser Begriff beschreibt die Fähigkeit mobiler Anwendungen, sich an verändernde Umgebungen anzupassen. Die Adaption kann *anwendungsbewußt* oder *anwendungstransparent* geschehen. Im anwendungstransparenten Fall wird z.B. durch den Einsatz eines zur Server-Kommunikationsschnittstelle kompatiblen Proxies auf dem Client, der mobile Betrieb von sonst in nicht-mobilen Umgebungen ausgeführten Anwendungen ermöglicht. Dieser Proxy *hortet* bei bestehender Internetverbindung zunächst benötigte Daten in einem lokalen Cache. Bei Verlust der Internetverbindung *emuliert* er die Funktionalität des Servers und merkt Änderungen von Daten lokal vor. Bei wiederhergestellter Internetverbindung *reintegriert* der Proxy die Änderungen in den Datenbestand des Servers, wobei eine Konfliktbehandlung nötig sein kann. Im anwendungsbewußten Fall wird eine Applikation über qualitative und quantitative Veränderungen der Verfügbarkeit von Ressourcen informiert, und passt ihre Anwendungslogik eigenständig an.

- *Extended Client-Server Model*: Dieses Modell beschreibt die Fähigkeit eines Client-Server-Systems, Funktionalität in Abhängigkeit von Umgebungsparametern dynamisch

zwischen Client und Server aufzuteilen, die bei klassischen Webanwendungen statisch zugeordnet ist. Ein Beispiel wäre die zeitweise Emulation von Serverfunktionen durch den Client in Abhängigkeit vom Internetverbindungsstatus oder seiner Qualität. In [ea99] wird z.B. als Generalisierung der Thin-Client-Architektur[1] und Full-Client-Architektur[2] die *Flexible Client-Server Architecture* vorgestellt, die dank Mobiler Objekte (enthalten Programmeinheiten wie z.B. Threads), Funktionalität zwischen Client und Server verlagern kann.

- *Mobile Data Access*: Die Art und Weise der Datenübermittlung vom Server, die Datenstruktur im mobilen Netzwerk und die Methoden zur Erhaltung der Konsistenz des Caches des Clients werden mit dem Begriff *Mobile Data Access* zusammengefasst. Daten können prinzipiell über zwei verschiedene Mechanismen vom Server an Clients übertragen werden, dem Server-Push[3]- oder dem Client-Pull[4]-Mechanismus. Herausforderungen im mobilen Einsatz bestehen im lokalen Precaching von Daten, die bei einer schlechter oder gar keiner Verbindung lokal verwendet werden können, sowie in der Reintegration von während der lokalen Emulation (s.o.) bearbeiteten Daten in den serverseitigen Datenbestand.

Zu weiterführender Information wird hier auf die Quelle "Client-Server Computing in Mobile Environments" (Jin Jing et al.) [ea99] verwiesen.

2.2 Location Based Services

LBS sind drahtlose IP-Services zur Versorgung mobiler Nutzer mit Informationen in bezug auf dessen Standort, d.h., "alle Anwendungs-Services, die die Position eines mobilen Terminals nutzen" (siehe [MM08]).

Eine typische Umgebung für die Benutzung eines LBS ist in Abbildung 2.1 dargestellt. Ein mobiler Client, der entweder über das Global Positioning System (GPS) oder eine Form der Netzwerkortung Informationen über seine derzeitige Position erhält, greift über ein Netzwerk, z.B. das Global System for Mobile Communications (GSM)-Mobilfunknetzwerk oder ein Wireless Local Area Network (LAN), auf den im Internet bereitgestellten Dienst eines LBS-Providers zu (z.B. das in dieser Arbeit vorgestellte MOSM). Dem LBS-Provider steht entweder eine eigene Grunddatenbasis für die Ausführung seiner Funktionen oder die eines Content-Providers, der ein normaler Webservice im Internet sein kann (z.B. OSM), zur Verfügung.

[1] Thin-Client: Die gesamte Funktionalität liegt beim Server, wobei der Client nur als Nutzerinface dient.
[2] Full-Client: Die Funktionalität des Servers wird vom Client emuliert, womit diese wesentlich weniger von der Verbindung zum Server abhängt.
[3] Server-Push: Der Server sendet kontinuierlich Daten an Clients, ohne dass diese eine Anfrage stellen (Broadcast).
[4] Client-Pull: Dem Senden von Daten an einen Client geht eine Anfrage des Clients voraus.

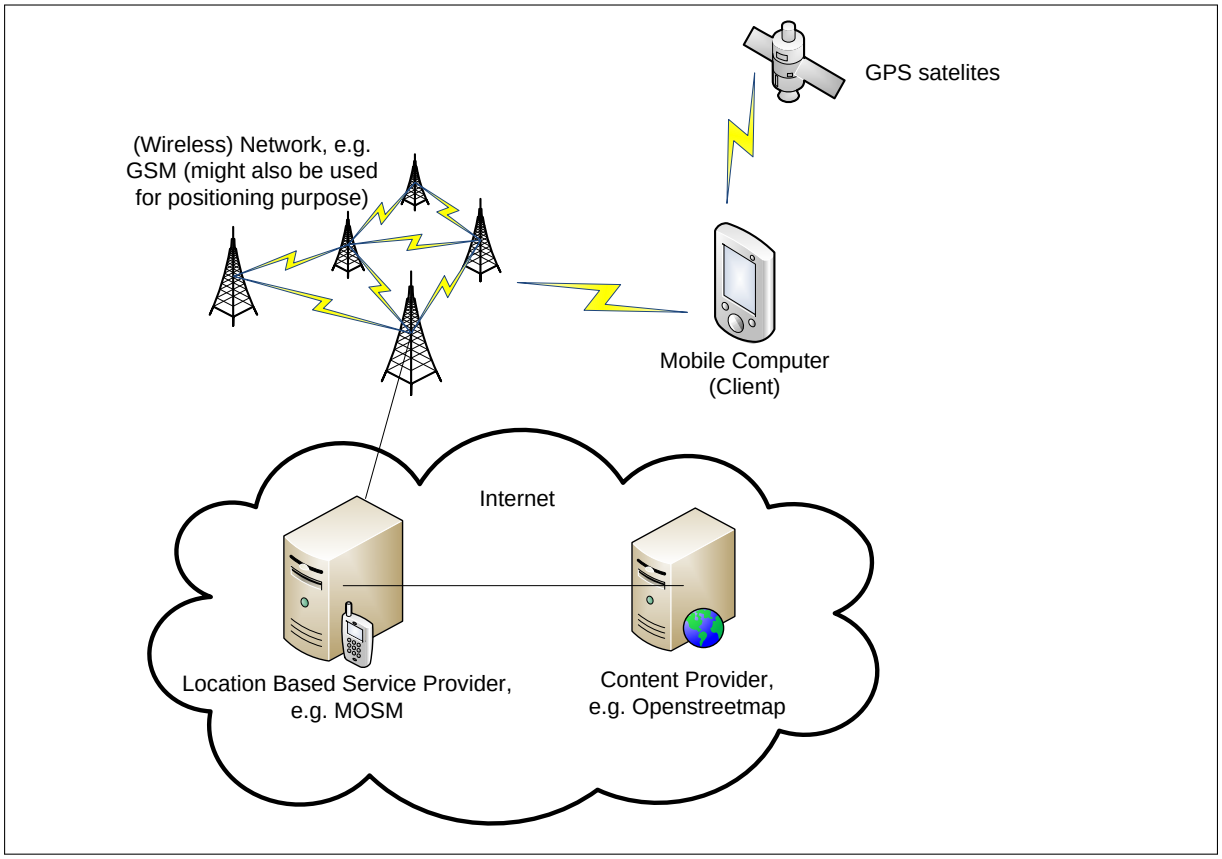


Abbildung 2.1: Anwendungsumgebung von Location Based Services

Ein LBS kann in Abhängigkeit von der Anwendungslogik des Dienstes sowohl als Pull-Service wie auch als Push-Service oder als eine Mischform aus beidem implementiert sein. Ein Pull-Service reagiert auf Anfragen des Clients und beantwortet diese entsprechend dem Inhalt der Anfrage. Ein Push-Server "schiebt" Daten unaufgefordert zum Client, sobald ein entsprechendes Ereignis stattgefunden hat (Trigger-Event). Dabei könnte es sich z.B. um standortbezogene Werbung oder Veranstaltungsinformationen handeln. In [GSM03] wird als dritte Form eines LBS der Tracking-Service angeführt. Dabei ist einem eingeschränkten und autorisierten Nutzerkreis die Information über die Position des betreffenen mobilen Clients zugänglich. Ein Beispiel für diese Form eines LBS ist ein "Buddy-Tracker", welcher die Position von Freunden in der Umgebung anzeigt.

In "Techniques for LBS Cartography" (siehe [car]) werden LBS nach dem Inhalt der von ihm kommunizierten Daten untergliedert. Es ergibt sich eine Einteilung in

1. **zweckbezogene LBS:** Hierbei handelt es sich um spezialisierte Anwendungen mit einem höchstwahrscheinlich sehr eingeschränkten Nutzerkreis (z.B. eine standortbezogene ärztliche Notrufapplikation oder ähnliches)

2. **allgemeine LBS:** Dies sind Services, die die Kernanwendungen bei der Nutzung von LBS bilden. Oft werden bei der Bearbeitung eines Requests durch LBS-Provider mehrere

dieser Kernanwendungen verwendet, um eine Antwort zu generieren. Bei der Suche nach einem Restaurant können z.B. die folgenden Kernanwendungen mitwirken:

- Positionsbestimmungsdienst (Gateway Service): Finden des aktuellen geografischen Standorts über "Network Positioning"
- standortbezogenes Adressbuch (Directory Service): Finden von Restaurants in der Umgebung des Standorts
- "Location Utility Service": Finden der Position des Restaurants anhand seiner Adresse oder umgekehrt (z.B. auch die Adresse des Standorts des Nutzers anhand der geografischen Position)
- Präsentationsdienst (Presentation Service): Anzeigen eines Fotos des Restaurants und Anzeigen des Restaurants auf der Karte
- Routenplaner (Route Service): Finden des Weges vom Standort des Clients zum ausgewählten Restaurant.

Diese fünf aufgezählten Dienste bilden die in der Open Geospatial Consortium (OGC) OpenLS-Spezifikation die "Core Services" [MM08].

2.2.1 Gängige Ortungsmethoden für Location Based Services

Die Nutzung von LBS setzt die Kenntnis des aktuellen Standorts des Clients voraus. Gängige und hier relevante Ortungsmethoden sind die Ortung via GPS und die Ortung in Funknetzwerken. Für den Offlinebetrieb einer Anwendung spielt dabei hauptsächlich GPS eine Rolle.

Ortung via GPS

Viele mobile Computer wie Notebooks, PDAs oder Smartphones besitzen heutzutage einen GPS-Empfänger, der entweder als internes Device oder extern z.B. über den Universal Serial Bus (USB) oder eine Bluetooth-Schnittstelle angeschlossen werden kann. Ermöglicht wird diese Form der Positionsbestimmung durch zur Zeit 31 GPS-Satelliten, die sich auf sechs verschiedenen, kreisförmigen Bahnen in einer Höhe von 20.200 km mit einer Bahnneigung von 55° relativ zur Äquatorialebene der Erde bewegen. Die Positionsbestimmung eines Nutzers geschieht über die Laufzeitmessung von Signalen von verschiedenen Satelliten zum GPS-Empfänger. Die Position der Satelliten und ihre genaue Umlaufzeit um die Erde (ca. 11h,58 min) sind zu jedem Zeitpunkt bekannt. Jeder Satellit besitzt eine Atomuhr und sendet kontinuierlich Zeitsignale. Ein zum Zeitpunkt t_0 gesendetes Zeitsignal von einem Satelliten A wird von einem GPS-Empfänger zum Zeitpunkt $t_0 + \Delta t_A$ empfangen. Bei Kenntnis der Atomzeit zum Empfangszeitpunkt wäre es dem Empfänger nun möglich, seine Position P_{Recv} zu bestimmen (vereinfachte Betrachtung des Problems in einer Dimension):

$$P_{Recv} = P_{Sat_A} \pm c \cdot \Delta t_A,$$

wobei P_{Sat_A} die Position des Satelliten A und c die Lichtgeschwindigkeit ist, mit der sich das Signal fortbewegt. In diesem Anwendungsfall löst sich "±" zu einem eindeutigen Vorzeichen auf, da vom Satelliten aus die Richtung zur Erdoberfläche vorgegeben ist. Da GPS-Empfänger im Regelfall keine Atomuhr besitzen, reicht auch in einer Dimension ein Satellit nicht aus, um eine exakte Positionsbestimmung vorzunehmen. Dafür ist ein zweiter Satellit notwendig, der ebenfalls ein genaues Zeitsignal sendet und dessen Position bekannt ist. Sei seine Position P_{Sat_B} und die Signallaufzeit seines Signals zum Empfänger Δt_B. Das Fehlen einer exakten

Uhr wird nun durch die Möglichkeit, das Zeitintervall $\Delta t_A - \Delta t_B$ zwischen dem Eintreffen zweier sich entsprechender Zeitsignale der beiden Satelliten zu messen, kompensiert. Wegen

$$P_{Recv} - P_{Sat_A} = \Delta t_A \cdot c \tag{2.1}$$
$$|P_{Sat_B} - P_{Sat_A}| = (\Delta t_A + \Delta t_B) \cdot c \tag{2.2}$$

ergibt sich nun für die Position des Empfängers:

$$P_{Recv} = \frac{(\Delta t_A - \Delta t_B) \cdot c + |P_{Sat_B} - P_{Sat_A}|}{2} + P_{Sat_A}.$$

Für den dreidimensionalen Anwendungsfall werden Zeitsignale von vier Satelliten mit bekannter Position benötigt. Dabei dient, wie im eindimensionalen Fall, ein Satellit als Zeitgeber während die restlichen Signale der Laufzeitmessung zur Triangulation der Position des Empfängers dienen. Für die Berechnung der Position erhält man somit ein homogenes Gleichungssystem mit vier Gleichungen und vier Unbekannten.

Funknetzwerk-basierte Positionsbestimmung

Die Ortung eines Clients in einem Funknetzwerk setzt die genaue Kenntnis der Standorte der Funkantennen des jeweiligen Netzwerks seitens des Clients voraus, so z.B. der Base Station Tranceivers bei einem GSM-Mobilfunknetzwerk oder der Hotspots bei einem 802.11 Wireless LAN. Um unter dieser Voraussetzung die Postion eines Clients zu bestimmen, gibt es mehrere Möglichkeiten, die sich an der Genauigkeit des Resultats und der Anzahl der an der Ortung beteiligten Funkantennen unterscheiden:

- **Ortung anhand der Cell of Origin (COA):** Als Cell of Origin (COA) wird im Allgemeinen die Funkzelle bezeichnet, deren Funkantenne die geringste Entfernung zum aktuellen Standort des Clients hat. Diese Antenne kann durch die von ihr ausgesendeten Beacon Pakete identifiziert werden. Da der Radius des Einzugsgebiets jeder Funkzelle im Netzwerk bekannt ist, kann der Standort des Clients auf ein Gebiet um die Funkantenne mit dem entsprechenden Radius festgelegt werden. Zusätzlich kann die Zeit, die ein Signal von der Funkantenne bis zum Client benötigt, verwendet werden, um den Radius weiter einzugrenzen. Praktisch ist diese zusätzliche Eingrenzung kaum möglich. Einerseits sind die Uhren der Client-Endgeräte zu ungenau, um die Entfernung einer Antenne mit Hilfe eines Funksignals über nur einige hundert Meter bis zu einigen Kilometern auszumessen, andererseits erreichen Funksignale ein Client-Endgerät im Allgemeinen nicht auf einem geraden Weg, sondern werden davor mehrmals an Objekten, wie z.B. Häusern, reflektiert.

- **Ortung anhand der Time Difference of Arrival (TDOA):** Bei Vorhandensein eines ausreichend genauen Synchronisierungsmechanismus von Uhren im Funknetzwerk, kann ein Client, ähnlich wie bei der GPS-Ortung, seine Position anhand von Zeitsignalen, die von den Funkantennen ausgesendet werden, bestimmen. Analog zur GPS-Ortung werden auch hier für eine zweidimensionale Ortung wenigstens drei Funkantennen und für eine dreidimensionale Ortung wenigstens vier Funkantennen benötigt, wobei in jedem Fall eine Antenne immer als Zeitgeber fungiert. Speziell in einem GSM-Netzwerk ist diese Anzahl von Antennen oft gleichzeitig verfügbar, um einerseits in dicht besiedelten Gebieten das hohe Datenaufkommen zu bewältigen und andererseits ein fehlerfreies

Handover sich bewegender Clients zwischen GSM-Zellen zu ermöglichen. Die Unschärfe dieses Ortungsverfahrens wird auch hier dadurch verursacht, dass nur selten eine direkte Sichtverbindung mit allen an der Ortung beteiligten Antennen besteht und somit eine Messung mit reflektierten Signalen stattfindet. Trotzdem ist diese Art der Positionsbestimmung genauer als die Ortung anhand der Cell of Origin, da die Triangulation des Clients seine Position enger eingrenzt als die bloße Kenntnis über die Funkzelle, in der er sich befindet.

- **Ortung anhand des Angle if Arrival (AOA):** Eine selten angewendete Ortungsmethode in Funknetzwerken funktioniert über die Messung des Winkels zu einer vorgegebenen Richtung in einem global festgelegten Koordinatensystem, unter dem das Funksignal die Antenne erreicht. Da für diese Form der Ortung spezielle Antennen notwendig sind und trotzdem auch hier eine Sichtverbindung zu den an der Messung beteiligen Antennen die Voraussetzung für ein korrektes Resultat sind, spielt sie gegenüber den zuvor genannten Methoden eine untergeordnete Rolle. Im Gegensatz zur Ortung über die TDOA übernimmt hier das Netzwerk die Berechnung der Position des Client-Endgeräts und nicht das Endgerät selbst.

Positionsbestimmung in Gears

Mit der Geolocation API liefert Gears ein Werkzeug, um die geografische Position eines Clients zu bestimmen. Die Geolocation-Klasse ist sowohl in der Lage über einen asynchronen Request die aktuelle Position eines Clients zu bestimmen[5] als auch dessen Position kontinuierlich zu überwachen[6]. Zur Positionsbestimmung kann die Geolocation-Klasse mehrere Quellen, sogenannte *Location-Providers*, nutzen, die sowohl lokal (z.B. bei Nutzung von GPS) als auch entfernt im Netzwerk (Netzwerkortung) liegen können. Die Kommunikation mit einem Location-Provider erfolgt über ein HTTP-POST mit einem als als JavaScript Object Notation (JSON)-formatierten Body seitens des Clients. Dieser Body enthält z.B. Informationen über sich in Reichweite befindliche GSM-Zellen oder Hotspots, anhand derer in diesem Fall eine Netzwerkortung möglich ist. Die Antwort des Location-Providers ist ebenfalls JSON-formatiert und liefert als Ergebnis die GPS-Koordinaten der Position des Clients sowie optional die Adressdaten, die zu dieser Position gehören. Die Genauigkeit der zweidimensionalen Ortung ist im Ergebnis über das Attribut *accuracy* und die der Höhenbestimmung über das Attribut *altitude_accuracy* angegeben.

2.3 Geografische Informationssysteme (GIS)

2.3.1 Definition

Als Geographisches Informationssystem (GIS) bezeichnet man Systeme, die der Eingabe, Speicherung, Bearbeitung, Abfrage, Analyse und Ausgabe von geografischen Daten dienen. Der Begriff umfasst sowohl Hardware als auch Software sowie die das System betreibende Organisation, ihr Personal, Informationen, auf deren Grundlage das System arbeitet, Kunden, die das fertige Produkt kaufen und nutzen und Anbieter, die Hardware und Software liefern

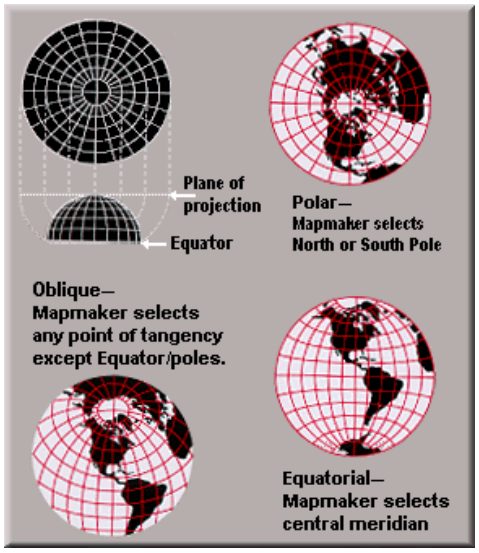
[5] Geolocation.getCurrentPosition(successCallback, [errorCallback], [options])
[6] Geolocation.watchPosition(successCallback, [errorCallback], [options])

[Dem09]. Im engeren Sinne wird hier die Softwarekomponente des Systems genauer betrachtet. Der hauptsächliche Zweck eines GIS liegt in der Analyse geografischer Objekte in bezug auf ihre Eigenschaften, ihre geografische Position und ihre Beziehung untereinander.

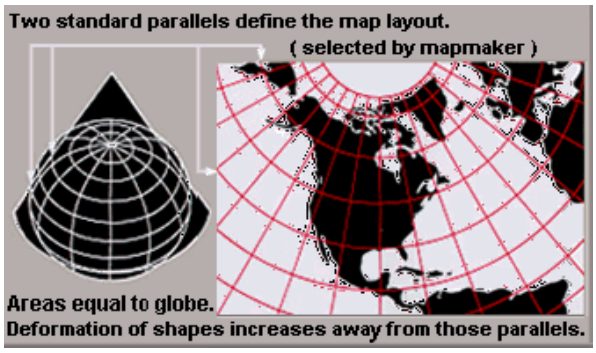
2.3.2 Darstellungskonzepte geographischer Daten

Kartenprojektionen

Es gibt drei verschiedene Grundtypen von Kartenprojektionen (siehe Abbildung 2.2), die sich auf die Projektion der Kugeloberfläche der Erde auf eine ebene Fläche, eine Zylinderinnenwand oder eine Kegelinnenwand zurückführen lassen. Eine gängige und häufig benutzte Form der Projektion bei der Erstellung von geografischen Karten ist die Mercator-Projektion. Sie wird standardmäßig auch von Openstreetmap zum Rendern von Karten mittels Mapnik oder Tiles@Home benutzt. Hierbei wird die Kugeloberfläche senkrecht zur Erdachse auf eine Zylinderinnenwand projektiert, wodurch eine winkeltreue Abbildung der Erdoberfläche erstellt wird. Das hat zur Folge, dass vom Äquator ausgehend sowohl in nördlicher als auch in südlicher Richtung eine immer stärker werdende Streckenverzerrung in Ost-West-Richtung auftritt. Nord- und Südpol selbst werden zu einer Geraden verzerrt, sind in der Projektion also keine eindeutigen Punkte mehr.

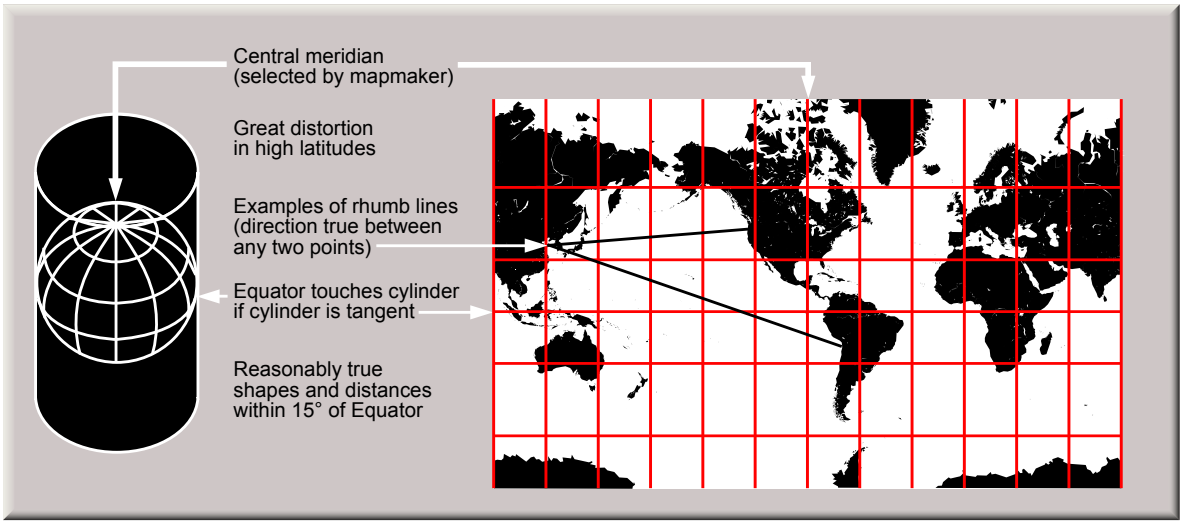


(a) Projektion an einer Fläche



(b) Projektion an einer Kegelinnenwand

Abbildung 2.2: Grundtypen von Kartenprojektionen



(c) Projektion an einer Zylinerinnenwand

Abbildung 2.2: Grundtypen von Kartenprojektionen (Fortsetzung)

Datenmodelle

Die Darstellung geografischer Gebiete ist in der Praxis durch zwei grundsätzlich verschiedene Methoden realisierbar. Die Rasterdarstellung teilt ein Gebiet in eine Reihe von Teilgebieten (Kacheln) und ordnet diesen über die jeweilige Kachel gemittelte Werte für geografische Größen zu, während die Vektordarstellung ein exaktes Modell der geografischen Umgebung aus Punkten, Wegen und Polygonen erstellt.

Vektordarstellung Die Darstellung geografischer Daten im Vektormodell beruht auf der Verknüpfung einzelner oder mehrerer Knoten (Vektoren) mit Attributen. Ein Knoten ist das kleinste in diesem Modell darstellbare geographische Objekt und kann für räumlich nicht ausgedehnte Objekte, wie z.B. für einen Point of Interest (POI) oder für einzelne Gebäude, stehen. Räumlich ausgedehnte Objekte, wie z.B. Straßen, Flüsse, Parks oder Wälder, werden durch eine Menge von Knoten dargestellt. Hierbei wird zwischen Objekten unterschieden, die als Linie dargestellt werden können, wie z.B. Straßen, Waldwege und Flüsse, sowie Objekten, die eine Flächenausdehnung besitzen. Erstere werden durch Aneinanderreihung von Knoten zu einer Linie dargestellt. Eine solche Linie wird im Folgenden auch als Weg bezeichnet. Knoten, die zu einem Weg gehören, werden auch Stützstellen des Weges genannt. Das Modell eines Wegverlaufs muss in Abhängigkeit von seiner Geradlinigkeit mit mehr (bei krummem Verlauf) oder weniger (bei geradlinigem Verlauf) Stützstellen versehen werden, um einen bestimmten Grad an Genauigkeit zu erreichen. Geografische Objekte mit einer Flächenausdehnung, z.B. Parks, Wälder und Seen, werden durch einen oder mehrere geschlossene Wege modelliert. Diese werden im Folgenden Polygone genannt. Zur Unterscheidung verschiedener Typen von Knoten, Wegen und Polygonen werden diese mit Attributen behaftet.

Auf diese Weise lassen sich Karten mit einer sehr genauen Ortsbestimmung diskreter Objekte bzw. ihrer Ausdehnung und ihrer Grenzen erstellen. Beziehungen zwischen geografischen Objekten sind in der Regel aus der Datendarstellung einfach ablesbar. So sind beispielsweise zwei Polygone benachbart, falls mindestens ein Wegstück existiert, das vollständig zur

Begrenzung beider Polygone gehört.

Systeme wie PostGIS[7] arbeiten mit der Vektordarstellung und verfügen über eine Vielfalt an Analysefunktionen. In Kapitel 4.2 werden einige dieser Funktionen exemplarisch vorgestellt. Diese Analysefunktionen sind in der Regel zwar genauer als ihre Äquivalente in der Rasterdarstellung, jedoch oft wesentlich rechenaufwendiger.

Rasterdarstellung Die Darstellung geografischer Daten im Rastermodell beruht auf der Aufgliederung eines geografischen Raumes in einzelne Kacheln, sodass diese zusammen die Gesamtoberfläche der Erde darstellen. Meist wird eine rechteckige bzw. quadratische Form der Kacheln gewählt, wobei jede Kachel dieselbe Anzahl von Längen- und Breitengraden abdeckt. Die einfachste denkbare Rasterdarstellung besteht aus einem zweidimensionalen Feld von Kacheln, wobei jeder Kachel ein Wert zugeordnet ist. Dieses Feld wird als Raster bezeichnet und bildet eine thematische Schicht. Auf diese Weise bietet die Rasterdarstellung kein exaktes Modell der Erdoberfläche, sondern stellt pro Kachel den Wert einer oder, bei Überlagerung von thematischen Schichten, mehrerer Eigenschaften für einen Repräsentanten der Punktmenge aus der Kachel dar. Durch die so entstehende Vergröberung von Strukturen sind viele analytische Berechnungen einfacher durchzuführen, da nun anstatt mit reellwertigen Koordinaten mit ganzzahligen Kachelindizes gerechnet wird.

Die Rasterdarstellung ist geeignet für die Repräsentation flächig kontinuierlicher Messgrößen (z.B. Geländeanstieg oder Bevölkerungsdichte). Dank der einfachen Datenstruktur sind einige algebraische Berechnungen bei der Benutzung des Rastermodells wesentlich weniger rechenintensiv als bei der Benutzung des Vektormodells. So ist z.B. die Flächenberechnung eines Polygons mit vielen Eckpunkten nach dem Vektormodell eine komplizierte Berechnung, während die gleiche Berechnung nach dem Rastermodell durch ein Zählen von Kacheln zu bewerkstelligen ist. Umgekehrt sind räumliche Transformationen, wie etwa eine Rotation eines Polygons um einen Punkt, im Vektormodell relativ einfach darzustellen. Es geschieht hier mit Hilfe einer einfachen Rotationsmatrix für die Koordinaten, während beim Rastermodel eine aufwendige Interpolation von Attributwerten der gedrehten Kacheln notwendig ist.

2.4 Technologien für den Betrieb von Offline-Webanwendungen

Gängige Technologien zur Erstellung von Online-Webanwendungen sind u.a.:

- das Hypertext Transfer Protocol (HTTP) als OSI-Schicht 5-7 Übertragungsprotokoll [R. 99]
- die Hypertext Markup Language (HTML) als gängigste Auszeichnungssprache für Webinhalte [Dav99]
- dynamisches HTML (JavaScript)
  (siehe European Computer Manufacturers Association (ECMA)-Spezifikation: [jse02],
  Mozilla JavaScript 1.5 Referenz: [jsm])
  und Document Object Model (DOM)[Arn04])
- serverseitiges Scripting, basierend auf

[7] PostGIS selbst ist kein geografisches Informationssystem im eigentlichen Sinne, sondern vielmehr eine Datenbankerweiterung, die die Nutzung der Datenbank "PostgreSQL" als GIS-Datenbank-Backend ermöglicht.

  - Server Side Includes (SSI) oder Active Server Pages (ASP), um z.B. PHP oder ASP.Net Skripte einzubinden, die serverseitig das zu übertragende Dokument verändern oder generieren
  - Common Gateway Interface (CGI), um Drittsoftware wie z.B. Perl-, Python- oder Tool Command Language (TCL)-Skripte oder C/C++-Programme einzubinden

- AJAX, das eine asynchrone Übertragung von Daten zwischen Client und Server ermöglicht, womit u.a. ein Neuladen einer Webseite bei Veränderung ihrer Daten unnötig wird. Dadurch wirkt eine Webanwendung dynamischer und nähert sich in ihrer Bedienung daher stark an Desktop-Anwendungen an.

In diesem Abschnitt werden Lösungen vorgestellt, die auf diesen Technologien aufbauen und den Betrieb von Webanwendungen trotz einer zeitweise getrennten Internetverbindung erlauben.

2.4.1 HTML5

Der sich zur Zeit noch in der Entwicklung befindende HTML5-Standard des World Wide Web Consortium (W3C) enthält Elemente zur Entwicklung von Offline-Webanwendung. So beinhaltet der HTML5-Standard die Möglichkeit, Daten in einer lokalen SQL-Datenbank abzuspeichern, zu lesen, zu bearbeiten und zu löschen. Um ein Einfrieren einer Applikation zu verhindern, sind SQL-Methodenaufrufe hier generell asynchron, d.h. nach Beendigung einer SQL-Anfrage wird ein eventuelles Ergebnis oder eine Fehlermeldung mittels einer Callback-Funktion an die Applikation zurückgegeben.

Ebenso enthält HTML5 ein Application Programming Interface (API), um Ressourcen von Webanwendungen offline verfügbar zu halten. In einem Manifest wird spezifiziert, welche Ressourcen für den Offlinebetrieb der Anwendung notwendig sind bzw. welche Ressourcen vom Caching ausgeschlossen werden, d.h. nur im Falle einer existierenden Internetverbindung zugänglich sein sollen.

Sowohl die oben beschriebene SQL-Funktionalität als auch das Caching von Ressourcen sind gleichermaßen in Gears (s.u.) enthalten.

HTML5 stellt weiterhin ein Navigator-Objekt bereit, welches über sein Navigator.onLine-Attribut Aufschluss über den derzeitigen Status einer Netzwerkverbindung gibt (online/offline). Das Navigation-Objekt unterscheidet dabei nicht zwischen Netzwerken, über die das Internet erreichbar ist, und abgeschlossenen Netzwerken. Somit ist das gesetzte onLine-Attribut des Navigator-Objekt bestenfalls ein Indiz für eine bestehende Internetverbindung, keinesfalls jedoch eine Garantie.

2.4.2 Gears

Zielsetzung

Gears ist eine unter BSD-Lizenz veröffentlichte Webbrowser-Erweiterung der Firma Google, die einen Offlinebetrieb von Webanwendungen ermöglichen soll. Die drei Hauptbestandteile, die Gears dafür zur Verfügung stellt sind

1. ein lokaler Server zum Bereitstellen von Anwendungsressourcen ohne Anfrage an den entfernten Server

2. eine lokale Datenbank, um Daten zwischenzuspeichern bzw. bereitzuhalten
3. ein Worker-Thread-Pool, der aufwendige Jobs im Hintergrund bearbeitet, ohne die Responsitivät der Anwendung zu beeinträchtigen

(siehe [geab])

Architektur

Gears bietet die Möglichkeit, Funktionalität einer Onlineanwendung auch offline verfügbar zu halten, ohne aber notwendigerweise den Gesamtumfang der Funktionalität einer Anwendung abzudecken. Verschiedene Ausbaustufen der Softwarearchitektur sind möglich, die sich untereinander in der Anzahl ihrer Abstraktionsschichten, ihrer Komplexität und der Behandlung von Veränderungen des Zustandes der Internetkonnektivität unterscheiden.

Herkömmliche Webanwendungen wie in Abbildung 2.2 kommunizieren direkt mit dem Server. Daher fehlt eine ausgesprochene Datenschicht, und die Funktionalität der Webanwendung ist ausschließlich bei Vorhandensein einer aktiven Verbindung zum Webservice-Anbieter gegeben.

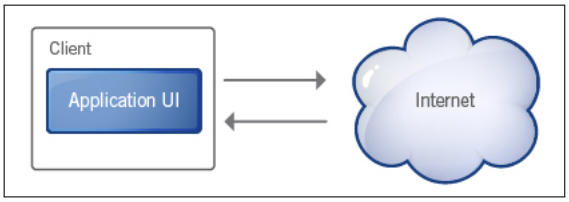


Abbildung 2.2: Webanwendung ohne Datenschicht

Zur strukturierteren Gestaltung einer Onlineanwendung ist die Einführung einer Datenschicht als zusätzliche Abstraktionsebene zwischen Anwendungsoberfläche und Web-Server sinnvoll. Jede Anfrage an den Server läuft somit über ein definiertes Objekt. Dieses entscheidet bei einer Anfrage nach einer Ressource, ob diese lokal geladen werden kann oder eine Anfrage an den Web-Server notwendig ist.

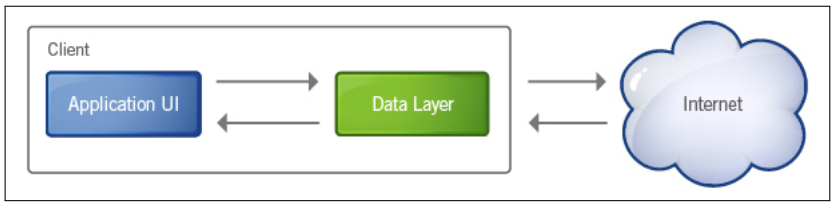


Abbildung 2.3: Webanwendung mit Datenschicht

Um eine Webanwendung teilweise oder vollständig offline verfügbar zu machen, wird die Architektur in Abbildung 2.2 zunächst um eine lokale Datenschicht ergänzt. Gears erlaubt es, in der Applikation zu definieren, welche Ressourcen lokal geladen werden dürfen und welche erzwungenermaßen immer neu vom Server geholt werden müssen. In der Architektur findet diese Entscheidung im “Data Switch” (siehe Abbildung 2.4) statt, das nach außen dasselbe Interface bereitstellt wie die Server-Datenschicht bzw. die neu dazugekommene lokale Datenschicht.

Um die lokale und entfernte Datenbank zu synchronisieren, sind zwei Ansätze möglich:

1. Manuelle Synchronisation zu einem vom Nutzer festgelegten Zeitpunkt: Dies erfordert, dass der Nutzer den tatsächlichen Verbindungsstatus zu dem von ihm festgelegten Synchronisationszeitpunkt kennt.

2. Automatische Synchronisation im Hintergrund der Anwendung: Diese Synchronisationsmethode kann zu einer Verschlechterung des Antwortverhaltens der Anwendung führen. Abhilfe kann die Verwendung des WorkerPools (siehe Abschnitt 2.4.2) schaffen, mit dessen Hilfe die Synchronisation im Hintergrund ausgeführt werden kann.

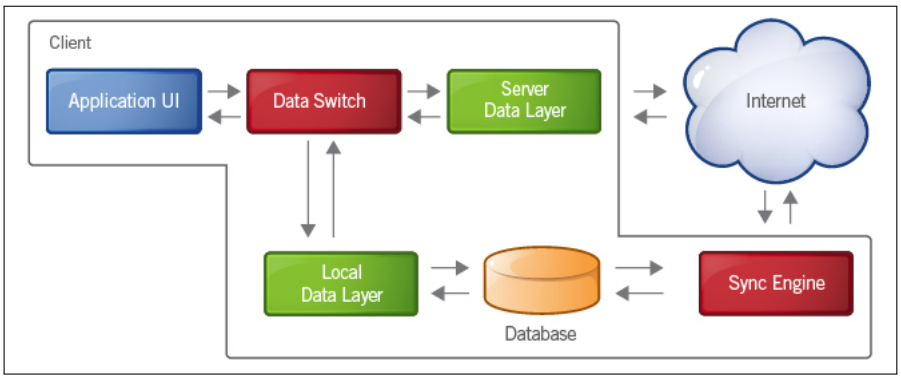


Abbildung 2.4: Webanwendung mit lokaler Datenschicht

API

Die Gears-API stellt folgende Module zur Verfügung:

- Factory: wird benutzt, um alle anderen Gears-Objekte zu instanziieren
- Blob, BlobBuilder: die Klasse Blob eröffnet die Möglichkeit, Binärdaten darzustellen; die Klasse BlobBuilder dient als Hilfsmittel, um Blobs zu erstellen
- Canvas: ist eine Bildbearbeitungs-API, die auch das Decodieren und Encodieren von Bildern in Formaten wie PNG oder JPG, die als Blobs vorliegen, erlaubt
- Database: stellt der JavaScript-Anwendung eine Browser-lokale relationale SQLite-Datenbank zur Verfügung
- Desktop: stellt desktopbezogene Funktionalität bereit, wie z.B. das Erstellen von Shortcuts
- Geolocation: stellt Funktionalität zur Bestimmung der geografischen Position eines Clients bereit. Dabei unterstützt die Klasse sowohl das Bestimmen der derzeitigen Position als auch die Beobachtung der zeitlichen Veränderung der Position und das Angeben der letzten bekannten Position
- HttpRequest: implementiert eine Untermenge der W3C XmlHttpRequest-Spezifikation (asynchroner AJAX-Request)

- LocalServer: stellt browser-lokale Methoden bereit, die HTTP-Ressourcen in einem lokalen Cache bereitstellen
- Timer: implementiert die WhatWG Timer-Spezifikation (Time Element in HTML5)
- WorkerPool: stellt Methoden bereit, um JavaScript Code im Hintergrund auszuführen, ohne das Antwortverhalten der Programmoberfläche zu beeinflussen

(siehe [geaa])

3 Design

In diesem Kapitel wird die Architektur des MOSM-Services beschrieben und begründet.

3.1 Openstreetmap-Komponenten

In Abbildung 3.1 sind die wichtigsten Openstreetmap-Komponenten sowie MOSM-Komponenten schematisch dargestellt. Das Openstreetmap-Backend besteht aus einer PostgreSQL-Datenbank, die über die HTTP-basierte OSM-API (derzeit in der Version 0.6) lesend ohne Authentifizierung und schreibend nach HTTP-Authentifizierung (HTTP Basic Authentication) oder OAuth-Authentifizierung zugänglich ist. Da die Datenbankstruktur des Backend-Servers im weiteren Verlauf dieser Arbeit nicht verwendet wird, wird an dieser Stelle für weiterführende Informationen auf das Openstreetmap-Wiki (wiki.openstreetmap.org/wiki/Database) verwiesen.

Für die grafische Umwandlung der Informationen aus der Datenbank ist das Rendering-System *Mapnik* zuständig. Hierfür wird ein aktueller mit dem Werkzeug *Osmosis* erstellter Dump der OSM-Datenbank in eine PostgreSQL mit PostGIS-Erweiterung (siehe 2.3, 4.2) importiert, die Mapnik als Datenquelle nutzt. Diese Datenbank wird regelmäßig mit Dumps der Backend-Datenbank bzw. mit inkrementellen Differenz-Dateien der Backend-Datenbank aktualisiert.

Durch die Konfigurierbarkeit mittels Style-Sheets ist Mapnik in der Lage, die Darstellungsform von geografischen Objekten auf der Karte auf vielfältige Art und Weise zu gestalten.

Ein weiteres Rendering-System ist das verteilt arbeitende *Tiles@Home*. Über einen zentral laufenden Server (*T@hngo*-Server) werden Anfragen zum Rendern von Kartenkacheln entgegengenommen und an Clients weitergeleitet, die das tatsächliche Rendern vornehmen und anschließend das Ergebnis zurück zum Server senden. Dieser stellt letztendlich die fertigen Kacheln öffentlich zum Download bereit. Als Rendering-Software wird hier *Osmarender* verwendet.

Einer von vielen möglichen Orte für die Darstellung der fertigen Kartenkacheln ist die sogenannte *Slippy Map* auf der OSM Homepage.

Die wichtigsten Komponenten der OSM-Architektur sind in Abbildung 3.1 grau unterlegt dargestellt. Die daran angeschlossenen Komponenten zur Ausführung von Openstreetmap als mobile Offline-Webanwendung sind darunter weiß unterlegt dargestellt. Diese Abbildung dient der groben Übersicht über die Openstreetmap-Komponenten und der Einordnung der vorliegenden Arbeit. Eine nährere Beschreibung zur erstellten Architektur und Anwendungslogik folgt in den Abschnitten 3.3 und 3.4.

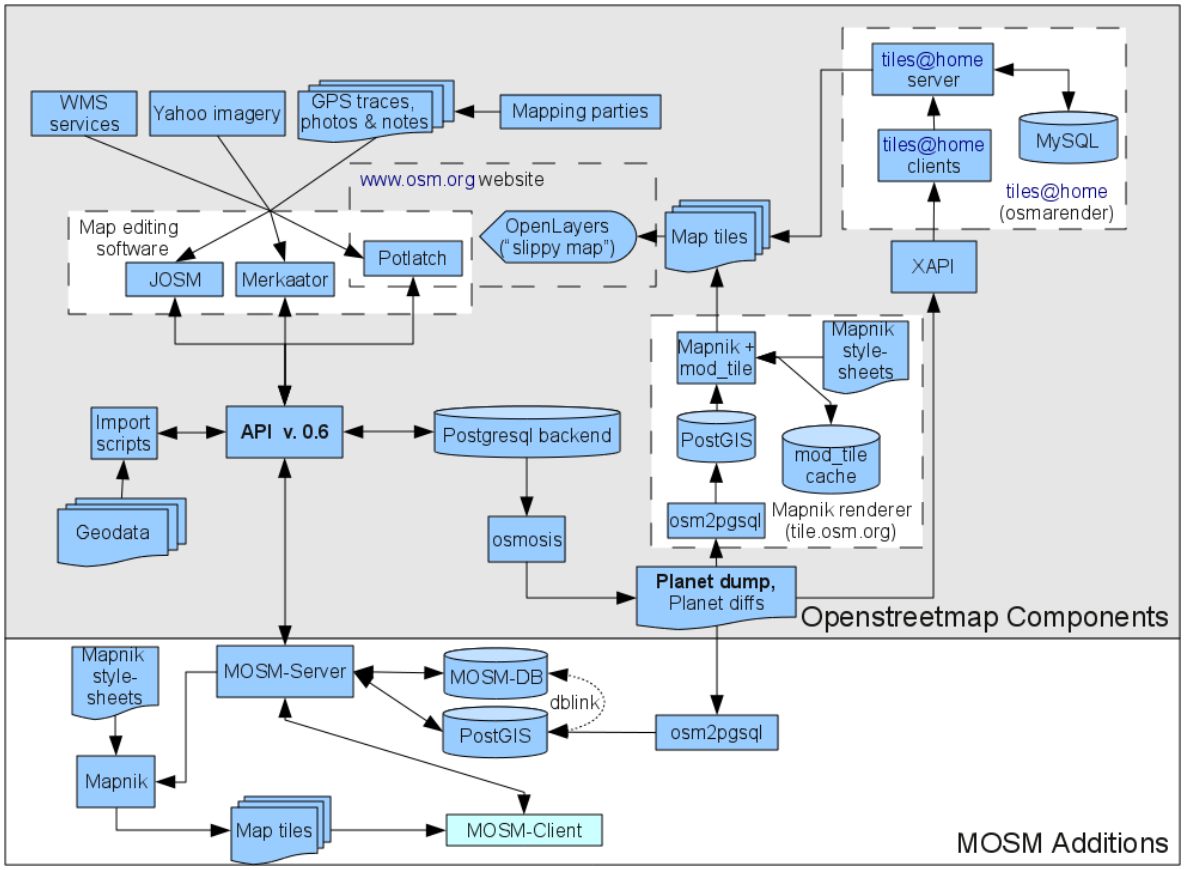


Abbildung 3.1: OSM- und MOSM Komponenten

3.2 Partitionierung der Funktionalität

Die Aufteilung der Funktionalität des Kartendienstes erfolgt für das hier vorgestellte Design statisch (siehe 2.1). Um die Laufzeit des Clients durch das Ausführen des Dienstes nicht über die Maßen zu verringern, werden rechenaufwändige Prozesse serverseitig ausgeführt. Dazu gehören Precaching-Berechnungen und das Rendern von Kartenmaterial. Die Frage, inwieweit ein eventuelles Rendering einer Teilmenge der Kartenkacheln auf dem Client die Nutzbarkeit des Dienstes verbessern könnte[1], wird im Rahmen dieser Arbeit nicht beleuchtet, soll aber als Anregung für eventuelle weiterführende Arbeiten hier erwähnt sein. Um die Funktionalität des Dienstes auch offline zu gewährleisten, gehört das Replizieren von Serverdaten (fertige Kartenkacheln und zum Bearbeiten der Karten notwendige Metainformationen), das Anzeigen und lokale Editieren (*Emulation*) und das Senden lokaler Änderungen zum Server (*Reintegration*) zum Funktionsumfang des Clients.

[1] Denkbar ist, dass die durch clientseitiges Rendern erfolgte Einsparung beim herunterzuladenen Datenvolumen einerseits die minimal benötigte Verbindungsdauer zum Server beim Replizieren von Daten senkt. Beim richtigen Verhältnis von herunterzuladenen Kartenkacheln zu selbst zu rendernden Kartenkacheln könnte die Laufzeit des Clients durch eine so erfolgte Energieeinsparung verlänger werden.

3.3 Clientseitiges Design

3.3.1 Anforderungen

Der Anforderungen an den Client lassen sich, wie folgt, untergliedern:

1. **Feststellen des Verbindungsstatus:** Um die Verfügbarkeit des Internetservice festzustellen, bedarf es eines Mechanismus, der den Internetverbindungsstatus überwacht und im Falle einer vorhandenen Verbindung das Ausführen von Requests startet.

2. **Darstellen und Bearbeiten lokal vorhandener Daten:** In der lokalen Datenbank gespeicherte Daten werden vom Client ausgewertet und dargestellt. Das Editieren von Daten geschieht zunächst lokal.

3. **Senden lokaler Änderungen:** Änderungen werden zu einem Zeitpunkt, an dem eine Verbindung zum Internet besteht, über einen Service-Request zum Server übertragen.

4. **Management eines Schreibcaches:** Lokale Veränderungen können im Fall einer nicht vorhandenen Internetverbindung nicht sofort ausgeführt werden und müssen somit bis zu einem Zeitpunkt, zu dem eine Internetverbindung existiert, vorgemerkt werden. Dazu dient ein Schreibcache, der einerseits das Vormerken realisiert und andererseits eventuelle Konflikte meldet (z.B. einen Konflikt, der durch das Vormerken sowohl einer Löschung eines Objektes als auch einer darauffolgenden Bearbeitung desselben Objektes entsteht).

5. **Sammeln von Bewegungsinformationen:** In der vorliegenden Arbeit wird angenommen, dass die Vorausschau auf die zukünftige Trajektorie eines Clients durch Auswertung des in OSM-Traces festgehaltenen Verhaltens vieler unkorrelierter Clients verbessert werden kann. Durch einen Rückgriff auf "alte Wege" sollte sich also die Genauigkeit der Vorhersage verbessern lassen. Zur Erhebung dieser Wege-Daten (Traces) sammelt und sendet ein Client Daten über seine letzten zurückgelegten Wege. Sich daraus ergebende, die Privatsphäre des Nutzers betreffende Probleme werden hier nicht betrachtet. Für einen produktiven Einsatz eines Systems, wie dem hier vorgestellten, ist deren Lösung allerdings unumgänglich.

6. **Verbindungsunabhängig erscheinende Funktionalität des Clients:**
   - **bei vorhandener Internetverbindung:**
     - **Replizieren von Serverdaten:** Da es sich bei den hier betrachteten Client-Endgeräten um mobile Geräte handelt, ist im Allgemeinen mit einer unvorhersehbar auftretenden und zeitlich begrenzten Internetverbindung zu rechnen. Um die Verfügbarkeit des Dienstes unter diesen Umständen gewährleisten zu können, müssen Daten, die in naher Zukunft benötigt werden, zunächst vom Server heruntergeladen und anschließend lokal gespeichert werden. Da der lokale Cache eines mobilen Clients sehr begrenzt ist, nach heutigen Maßstäben einige hundert Megabytes bis zu einigen Gigabytes, ist auch ein Mechanismus notwendig, der lange nicht benötigte Daten zugunsten neu angefragter Daten verdrängt.
     - **Abarbeiten von Einträgen aus dem Schreibcache:** Lokal vorgenommene Änderungen werden unter Nutzung einer API zum Server übertragen. Dabei

können unter Umständen Konflikte auftreten, die auf im Verlaufe der Offline-Zeit auf von anderen Nutzern vorgenommene Veränderungen zurückzuführen sind. Diese Konflikte müssen entweder manuell durch Nutzerinteraktion oder automatisch aufgelöst werden.

- **bei nicht vorhandener Internetverbindung:**
  - Beim plötzlichen Abbruch einer Internetverbindung müssen Einträge nicht vollständig übertragener Schreibvorgänge im Schreibcache wiederhergestellt bzw. markiert werden. Nach dem Wiederherstellen der Verbindung ist eine Überprüfung des serverseitigen Status der Bearbeitung notwendig. Diese Überprüfung kann z.B. über eine Versionsnummer des betreffenden Objekts geschehen.

3.3.2 Anwendungslogik

Die Anwendungslogik des Clients wird mit Hilfe des folgenden Ablaufszenarios erläutert:

- **Zustand 1:** Der Client befinde sich im Initialzustand an einer geografischen Position (lon, lat)[2], d.h., er besitzt keine lokalen Daten.
- **Zustand 2:** In Reichweite eines Hotspots erlangt der Client eine Verbindung zum Internet.
- **Zustand 3:** Die Internetverbindung ist nach dem vollständigen Herunterladen von benötigtem Kartenmaterial zusammengebrochen.
- **Zustand 4:** Inzwischen hat sich die geografische Position des Clients geändert. Es besteht erneut eine Verbindung zum Internet über einen anderen Hotspot.
- **Zustand 5:** Nach einem erneuten Zusammenbruch der Internetverbindung ist diese wiederhergestellt. Die Client hat inzwischen seinen geografischen Standort ein weiteres Mal geändert.

Erläuterung zur Anwendungslogik:

- **Zustand 1:** In diesem Zustand stellt der Client, bis auf das Sammeln von Bewegungsinformationen, die für das Funktionieren einiger der später erklärten Precachingverfahren wichtig sind, keine Funktionalität bereit.
- **Zustand 2:** Da noch keine lokalen Daten vorhanden sind, wird ein Service-Request an den nun erreichbaren MOSM-Server gestellt, der Kartenmaterial anfordert. Dieser wird im Folgenden *Map-Request* genannt. Der Ablauf des Stellens und Beantwortens eines Service-Requests ist in Abbildung 3.2 als Sequenzdiagramm veranschaulicht. Nach dem Senden des Map-Requests werden, falls für das serverseitig angewendete Precachingverfahren notwendig, gesammelte Bewegungsinformationen gesendet.

[2] lon=Längengrad (longitude); lat=Breitengrad (latitude)

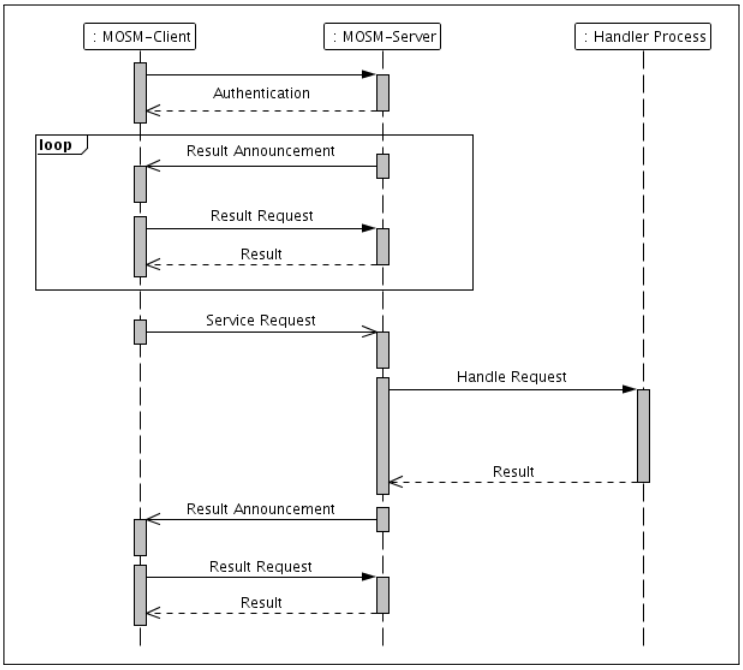


Abbildung 3.2: MOSM Client-Server Kommunikation, Behandlung eines Requests

Ein in eine Datensitzung eingebetteter Service-Request besteht aus folgenden Teilen:

- **Anmelden:** Eine Neuinitialisierung einer Sitzung beginnt mit der Authentifizierung des Clients am Server. Eventuell noch nicht abgeholte Ergebnispakete einer früheren Sitzung können nun dem sich neu angemeldeten Client zugeordnet und übermittelt werden. Falls der Server durch den Nutzer autorisiert wurde (OAuth: siehe 4.3.5), um auf die OSM-Datenbank zugreifen zu können, werden nun durch den Client lokal vorgenommene Änderungen zum OSM-Server übertragen.
- **Abfragen der serverseitigen Antwort-Queue:** Nach der Authentifizierung sendet der Server zunächst eine Ergebnisankündigung mit Einträgen von Ergebnispaketen früherer Service-Requests, die vom Server noch nicht erfolgreich zum Client übermittelt werden konnten. Die Liste der Einträge kann in dem Fall, dass alle bisherigen Pakete vergangener Sitzungen erfolgreich übermittelt wurden, auch leer sein. Der Client entscheidet nun, welche der angekündigten Ergebnisteile noch relevant sind und sendet einen entsprechenden Request zum MOSM-Server. Der angeforderte Teil des Ergebnisses wird daraufhin übermittelt, der Rest verworfen.
- **Senden des eigentlichen Service-Requests:** Abhängig von der geografischen Position und der Hauptbewegungsrichtung[3] des Clients wird ein Service-Request erstellt, der Kartenmaterial (Kartenkacheln und Metadaten, die zum Editieren der Karte benötigt werden) passend zu den übergebenen Parametern anfordert, eine lokale Änderung auf dem Server ausführen oder eine Suche nach geografischen Objekten initiieren soll. Das Senden dieses Requests und das darauffolgende

[3]Die Hauptbewegungsrichtung ergibt sich aus der Differenz der aktuellen Koordinaten und den Koordinaten, die sich in einer festgelegten Entfernung entlang der zurückgelegten Wegtrajektorie hinter den aktuellen Koordinaten befindet. Reicht die bisher zurückgelegte Entfernung nicht aus um auf diese Weise die Hauptbewegungsrichtung zu ermitteln, wird diese auf $(0, 0)$ gesetzt.

Empfangen der Ergebnisankündigung erfolgen asynchron, um die serverseitige Anwendungslogik vom Verbindungsstatus des Clients unabhängig zu halten.

  - **Empfangen der Ergebnis-Ankündigung:** Dem Senden der eigentlichen Ergebnisdaten geht das Senden einer Ankündigung voran, die Metadaten zu den serverseitig vorliegenden Ergebnisdaten enthält. Anhand dieser Daten entscheidet der Client, welche Teile des Ergebnisses noch benötigt werden bzw. lokal bereits vorhanden sind. Mit einem entsprechenden Request wird die Übertragung der Ergebnisdaten gestartet. Nicht angeforderte Ergebnisteile werden serverseitig verworfen.
  - **Empfangen der Ergebnisdaten:** Empfangene Ergebnisdaten werden entsprechend der mit ihnen verknüpften Metadaten in die lokale Datenbank des Clients geschrieben und können nun dargestellt bzw. editiert werden.

- **Zustand 3:** Die jetzt lokal vorhandenen Kartenkacheln und Metadaten können nun offline verwendet werden. Über ein Nutzerinferface werden die Kartenkacheln, die im lokalen Cache bevorratet sind, angezeigt und die Metadaten z.B. als thematische Schicht darübergelegt (so könnten z.B. POIs eines Gebietes als thematische Schicht über den Kartenkacheln angezeigt werden). Das Nutzerinterface erlaubt ebenfalls den Gegebenheiten des Client-Gerätes angepasste Bearbeitungen der lokalen Daten (d.h. an Gegebenheiten wie z.B. die Displaygröße). Solange keine Internetverbindung vorhanden ist, werden diese Änderungen in einem Schreibcache vorgemerkt und zum Server übertragen, sobald eine erneute Internetverbindung besteht. Zur Funktionalität des Clients gehört die Auflösung von eventuell auftretenden Konfliken beim Bearbeiten des Kartenmaterials (z.B. das Verhindern der Ausführung des Updates eines Knoten, der zuvor lokal gelöscht wurde). Andere schreibende Anfragen, die sich auf dasselbe geografische Objekt beziehen, werden zur Einsparung von späteren Service-Requests nach folgendem Schema zusammengefasst:

$$\begin{aligned} create + update &\rightarrow create \\ create + delete &\rightarrow \text{NULL} \\ update + update &\rightarrow update \\ update + delete &\rightarrow delete \end{aligned}$$

- **Zustand 4:** Es wurden diverse Änderungen am lokalen Datenbestand des Clients vorgenommen (z.B. durch das Hinzufügen diverser POIs). Weiterhin hat sich die geografische Position des Clients geändert, wodurch nun neues Kartenmaterial benötigt wird, das die neue Umgebung abbildet. Um die zukünftige Funktionalität des Clients nach einem erneuten Zusammenbrechen der Internetverbindung zu gewährleisten, werden zunächst lesende Operationen ausgeführt (z.B. das Holen von Kartenmaterial und das Stellen von Suchanfragen, z.B. nach Restaurants in der Umgebung). Die Beantwortung dieser Anfragen erfolgt ansynchron, womit der Schreibcache direkt nach dem Stellen der lesenden Anfragen abgearbeitet werden kann. Jede dieser Anfragen erfolgt nach dem oben bereits beschriebenen Sequenzdiagramm in Abbildung 3.2. Beim hier betrachteten Fall breche die Internetverbindung zusammen, während sowohl Requests zur Übertragung der Einträge im Schreibcache noch nicht vollständig abgearbeitet wurden als auch die Ergebnisübertragung zum Client noch nicht beendet war. Somit werden zunächst sämtliche noch nicht oder unvollständig übertragene Einträge im Schreibcache gehalten.

- **Zustand 5:** Nach der Authentifikation am MOSM-Server (siehe Sequenzdiagramm in Abbildung 3.2) empfängt der Client eine nicht-leere Ergebnisankündigung vom Server, da verschiedene Ergebnisse aufgrund der letzten zusammengebrochenen Internetverbindung nicht übermittelt werden konnten. Abhängig von der aktuellen Relevanz der angekündigten Ergebnisse sendet der Client den passenden *Result-Request*, woraufhin der Server angeforderte Ergebnisse zum Client übermittelt und den Rest verwirft. Die Relevanz alter Ergebnisse lesender Anfragen ergibt sich aus der derzeitigen Entfernung zu dem Gebiet, auf das sich das Ergebnis bezieht. Statusmeldungen schreibender Anfragen sind unabhängig von dieser Entfernung stets relevant.

  Nach dem Abarbeiten des Schreibcaches stellt der Client einen Map-Request (s.o.) für den aktuellen Standort. Da der lokale Cache des Clients eine begrenzte Größe hat, müssen ab einem gewissen Zeitpunkt Kartenkacheln aus dem Cache verdrängt werden, um Ergebnisse von aktuellen Anfragen speichern zu können. Dies sei nun der Fall. Daten bezüglich weit vom aktuellen Standort des Clients entfernter Gebiete wurden wahrscheinlich in der Vergangenheit selten oder überhaupt nicht verwendet. Durch die große Entfernung zwischen diesem Gebiet und dem derzeitigen Standort des Clients ist es ebenso wahrscheinlich, dass auf diese Daten auch in näherer Zukunft nicht zugegriffen werden muss. Sie können deshalb zugunsten neu angefragter Daten gelöscht werden. Um die aus der Größenbeschränkung des lokalen Speichers des Clients resultierenden Anforderungen (siehe 3.3.1) zu erfüllen, kann somit der Least Recently Used (LRU)-Algorithmus für das Caching von Daten verwendet werden.

  Abschließend wird erneut versucht, den Schreibcache abzuarbeiten.

Feststellen des Verbindungsstatus

Um den aktuellen Verbindungsstatus festzustellen, gibt es unterschiedliche, mehr oder weniger zuverlässige Ansätze. Der einfachste besteht darin, dem Nutzer die Entscheidung zu überlassen, ob eine Verbindung zum Internet besteht. Diese Verlagerung der Verantwortlichkeit zum Nutzer birgt allerdings die Gefahr in sich, dass dieser eventuell kein Wissen über den tatsächlichen Verbindungsstatus zum Internet besitzt und daher falsche Informationen über den Internetverbindungsstatus von der Anwendungslogik abgefangen werden müssen. HTML5 liefert mit dem Navigator-Objekt ein Werkzeug, um die Verbindung zu einem Netzwerk zu detektieren. Dies garantiert jedoch noch keine Verbindung zum Internet. Deshalb ist ein zusätzlicher Mechanismus notwendig, um die Information über den Verbindungsstatus abzusichern, wie z.B. ein Ping[4] zu einer bekannten URL. Um ein eventuelles DNS-Spoofing zu verhindern, sollten bei der anschließenden Nutzung von Webservices bzw. LBS mittels Public Key Infrastructure (PKI) die Identität und somit Vertrauenswürdigkeit des internetseitigen Kommunikationspartners überprüft werden (z.B. über das SSL Handshake Protocol). Die in [off04] vorgestellte Architektur führt als ein Anwendungselement einer Offlineanwendung eine ConnectionManager-Komponente ein. Indem für diese Komponente das *IConnectionDetectionStrategy-Interface* implementiert wird, wird die Anwendung in die Lage versetzt, den Internetverbindungsstatus automatisch erkennen zu können. Die Kriterien für das Vorhandensein einer Internetverbindung werden letztendlich auch hier vom Anwendungsprogrammierer vorgegeben.

[4]Ping=ICMP Echo Request; Das Ausbleiben eines ICMP Reply bedeutet nicht zwangsläufig, dass keine Verbindung zum Internet besteht, sondern eventuell nur, dass diese durch eine Firewall verworfen werden.

Um ebenfalls die Qualität der Internetverbindung, d.h., die verfügbare Datenrate, in die Anwendungslogik einzubeziehen, könnte bei der vorliegenden mobilen Openstreetmap-Anwendung die Anzahl der zurückzuliefernden Kartenkacheln nach einer Anfrage abhängig von der verfügbaren Datenrate gemacht werden. Dies wäre z.B. realisierbar durch die Einschränkung der Zoomstufen, in denen die Ergebniskacheln geliefert werden.

3.3.3 Resultierende Architektur des Clients

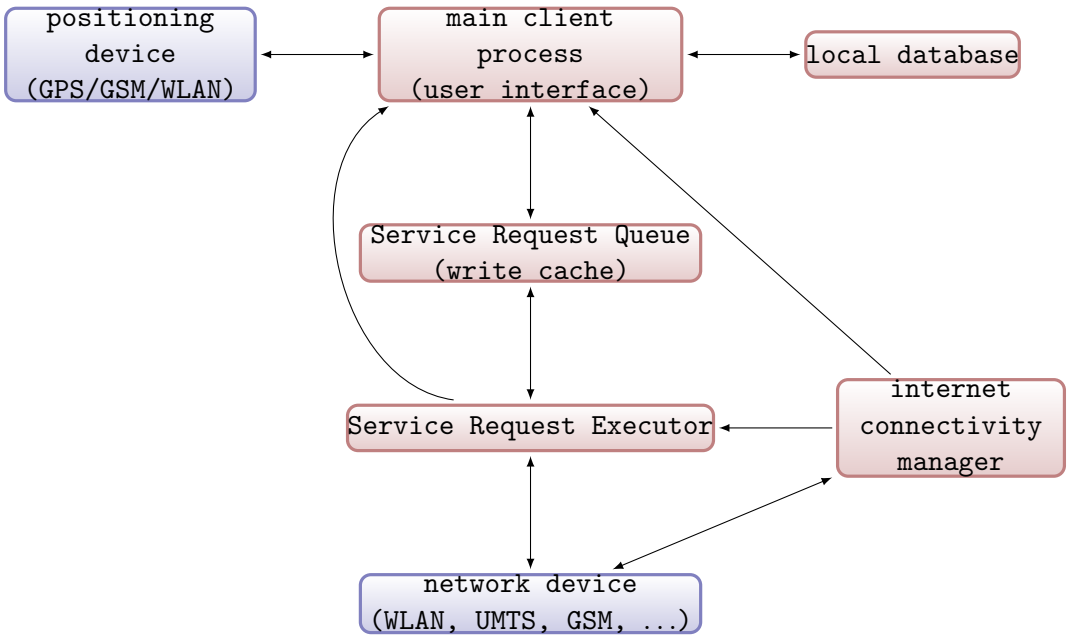


Abbildung 3.3: Architektur des MOSM-Clients

Die in Abbildung 3.3 dargestellte Architektur enthält die Komponenten, welche zur Umsetzung der o.g. Anforderungen und der gewünschten Anwendungslogik benötigt werden und folgende Aufgaben übernehmen:

- **main client process:**
  - Darstellen lokal vorhandener Daten
  - Erstellen und Verwalten von Service-Requests sowie Abfangen von Konflikten
  - Einfügen und Löschen von Ergebnisdaten der Service-Requests nach Least Recently Used (LRU) in die lokale Datenbank
  - Sammeln von Bewegungsinformationen für die serverseitige Statistik
- **positioning device:**
  - physikalisches Device zum Bereitstellen von geographischen Postions- und Bewegungsinformationen (aktuelle Position, Bewegungsrichtung, Geschwindigkeit)
- **internet connectivity manager:**
  - Feststellen des aktuellen Internetverbindungsstatus

- **local database:**
  - Speichern von Kartenmaterial und Metainformationen
- **service request queue:**
  - Verwaltung von geplanten Service-Requests im Offline-Modus
- **service request executor:**
  - Ausführen von Service-Requests und Weiterleiten des Ergebnisses an den "main client process"
  - Fehlerbehandlung bei plötzlicher Änderung des Verbindungsstatus und Wiederherstellung der "service request queue"
- **network device:**
  - physikalisches Device zur Herstellung einer Internetverbindung

3.4 Serverseitiges Design

3.4.1 Anforderungen

Die Anforderungen an den Server können wie folgt untergliedert werden:

1. **Bereitstellen einer API, die asynchrone Anfragen unterstützt um folgende Funktionen auszuführen:**
   - Bearbeiten von Anfragen nach OSM-Kartenmaterial in Abhängigkeit von Position und Bewegungsrichtung des anfragenden Clients
   - Bereitstellen von Methoden, um, autorisiert durch den Nutzer, lesend und schreibend über die OSM-API auf die OSM-Datenbank zugreifen zu können

   Im Falle des Zusammenbrechens der Verbindung zwischen Client und Server nach dem Stellen eines Requests soll die Anwendungslogik des Servers für das Bearbeiten des Requests davon unabhängig bleiben. Bei einem erneuten Verbinden des Clients mit dem Server besteht dann die Möglichkeit, das fertige Ergebnis herunterzuladen oder zu verwerfen.

2. **Liefern einer guten Ergebnisqualität:** Hauptziel der Arbeit ist es, bei einem gegebenen Request ein Ergebnis zu liefern, das möglichst genau dem gerade benötigten Kartenmaterial entspricht. Dieses Ziel könnte durch schlichtes Vergrößern der Ergebnismenge (Kartenmaterial) erreicht werden. Dies stünde jedoch in Konflikt mit einer potentiell teuren und/oder langsamen, auf jeden Fall jedoch unvorhersehbar instabilen Internetanbindung des Clients. Daher besteht das Ziel in der Optimierung der Auswahl des Bereichs, für den Kartenmaterial zum Client übermittelt wird, bei gleichbleibender Gesamtgröße des ausgewählten Bereichs.

3.4.2 Anwendungslogik

Nach der Anmeldung eines Clients und einer eventuell notwendigen Autorisierung des Servers für den Zugriff auf geschützte OSM-Daten via OAuth (siehe 4.3.5) wird zunächst eine vom

Client erwartete Ergebnisankündigung gesendet, die entweder die leere Menge von Ergebnissen oder tatsächlich noch nicht ausgelieferte Ergebnisse aus der "Delivery Queue" ankündigt. In dem zuletzt genanntem Fall wird als Antwort eine Ergebnisanfrage erwartet, die das Ergebnis insgesamt oder in Teilen anfordert bzw. verwirft. Dementsprechend werden anschließend die Ergebnisdaten übermittelt und bei erfolgreicher Übermittlung die Einträge aus der "Delivery Queue" entfernt. Abhängig von der Art des Requests des Clients treten zwei Fälle auf:

1. Beim eintreffenden Service-Request handelt es sich um einen *Map-Request*:
   - Mit den Parametern des Requests, d.h., Position und Bewegungsrichtung des Clients, wird die Precaching-Methode gerufen, die die Metainformationen einer Menge von Kartenkacheln zurückliefert. Die Bilddateien der so referenzierten Dateien sind nun entweder auf dem Server bereits vorhanden oder können wahlweise über eine Mapnik-Instanz "gerendert" oder über den OSM-Tile-Service oder über Tiles@Home heruntergeladen werden. Die somit vollständig vorliegenden Ergebnisdaten werden dem Client nun über ein entsprechendes Paket angekündigt, das zunächst in die "Delivery-Queue" geschoben wird, bevor es gesendet wird. Nach dem Empfang der Ergebnisanfrage des Clients werden die angeforderten Ergebnispakete gesendet und der Rest wird verworfen.
2. Beim eintreffenden Service-Request handelt es sich um einen schreibenden Service-Request:
   - Hierbei handelt es sich um einen Request, der nicht direkt vom MOSM-Server behandelt wird, sondern über die OSM-API bzw. die
     OSM Extended API (OSMXAPI) zur Bearbeitung weitergeleitet wird. Falls der vom Client gestellte Request eine schreibende Operation auf der OSM-Datenbank erfordert, muss der Service-Request nach den OAuth-Vorgaben korrekt signiert sein (siehe 4.3.5). Im Falle eines lesenden Zugriffs auf die OSM-Datenbank (mit Ausnahme des Lesens von Benutzerinformationen) ist eine Signatur der Pakete nicht notwendig. Hier kann neben der OSM-API auch die OSM Extended API (OSMXAPI) (siehe 4.3.4) verwendet werden, die feingranulare Suchoptionen für lesende Anfragen unterstützt. Falls die ausgeführte Operation ein Ergebnis liefert, wird analog zum Prozedere beim "Map-Request" über die "Delivery-Queue" eine Ergebnisankündigung, und entsprechend der folgenden Ergebnisanfrage die Ergebnisdaten gesendet.

Bei Vorhandensein einer aktiven Verbindung zu einem Client wird eine Ergebnisankündigung aus der "Delivery-Queue" zum entsprechenden Client gesendet. Bei der anschließenden erfolgreichen Übertragung der Ergebnisdaten zum Client wird die dazugehörige Ergebnisankündigung in der "Delivery Queue" gelöscht. Bei einer unvollständigen Übertragung der Ergebnisdaten wird die Ergebnisankündigung nur um jene Einträge verkürzt, deren zugehörige Daten erfolgreich gesendet wurden. Bei einer erneuten Verbindung des betreffenden Clients wird nach der Authentifizierung die entsprechend verkürzte Ergebnisankündigung neu gesendet und eine erneute Übertragung der Ergebnisdaten versucht.

3.4.3 Precachingstrategien

In diesem Abschnitt werden verschiedene geografische Precachingmethoden vorgestellt, die in geometrische und nicht-geometrische Verfahren unterteilt werden können. Sämtliche hier

vorgestellten Verfahren arbeiten unter Verwendung des Rastermodells (siehe 2.2), womit die Ergebnismenge eines Precachingverfahrens immer aus einer Menge von Kartenkacheln besteht. Die Mächtigkeit dieser Menge ist für alle Precachingverfahren konstant gesetzt, um eine Vergleichbarkeit zwischen den verschiedenen Verfahren zu gewährleisten. Geometrische Verfahren hängen ausschließlich von Parametern ab, die durch den einzelnen Nutzer vollständig gegeben sind und beschränken sich hier auf die Position und die derzeitige Hauptbewegungsrichtung des Clients. Nicht-geometrische Verfahren beziehen das frühere Verhalten von Clients in die Berechnung der Ergebnismenge mit ein.

Geometrische Verfahren

Kreisförmiges Precaching Beim kreisförmigen Precaching handelt es sich um die einfachste denkbare Caching-Strategie. Alle Kartenkacheln in einem bestimmten Radius um den aktuellen Standort des Clients werden als Ergebnis zurückgeliefert. Dieses Verfahren liefert über die Zeit immer dasselbe Ergebnis und benötigt keine weiteren Metainformationen zu Erstellung der Ergebnismenge.

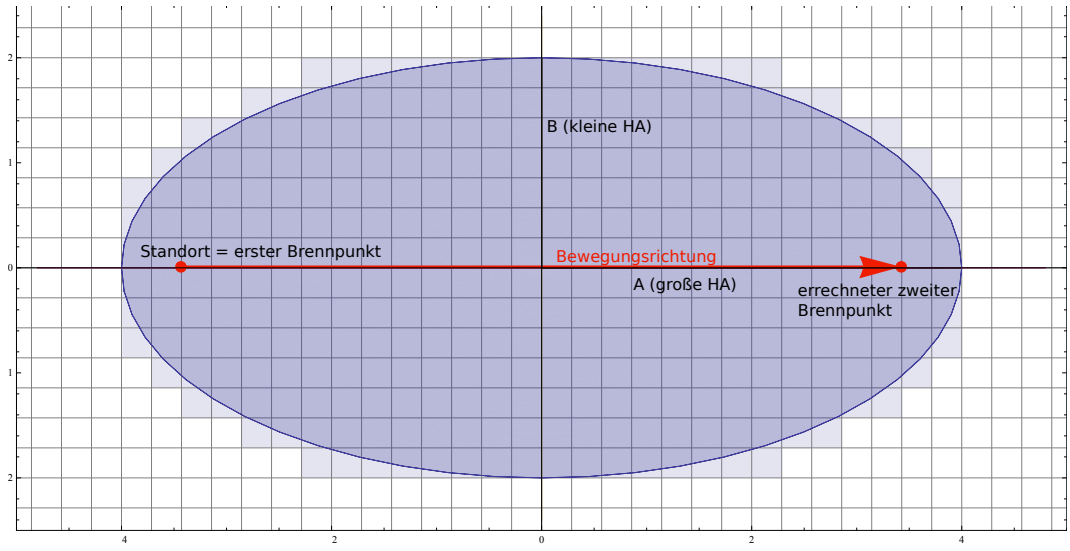


Abbildung 3.4: Geometrisches Cachingverfahren: Elliptisches Caching

Elliptisches Preaching Das elliptische Caching ist im Vergleich zum kreisförmigen Caching zusätzlich abhängig von der Bewegungsrichtung. Dazu wird eine Ellipse mit den Brennpunkten

$$\vec{P_1} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$$

(der Position des Clients) und

$$\vec{P_2} = \vec{P_1} + 2E \cdot \vec{e} = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix},$$

konstruiert, wobei $\vec{e}$ der Einheitsvektor in der momentanen Bewegungsrichtung ($\sim$ Geschwindigkeit) ist (siehe Abbildung 3.4 bzw. Abbildung 7.2 im Anhang).

Die kartesischen Koordinaten lassen sich für einen nicht zu großen Abstand zwischen beiden Brennpunkten mit Längen- und Breitengrad identifizieren:

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} \approx \begin{pmatrix} lon_{\vec{P}_i} \\ lat_{\vec{P}_i} \end{pmatrix}$$

Bei fester Zoomstufe werden diese Koordinaten in Längen- und Breitengradrichtung im Weiteren als zweidimensionale ganzzahlige Kachelindizes $(x, y) \in (\mathbb{N}^0)^2$ verstanden.

Wenn $\vec{P}_K = \begin{pmatrix} x_K \\ y_K \end{pmatrix}$ die Position einer (hinreichend kleinen) Kartenkachel ist, dann ist die Menge aller in das elliptische Precaching eingeschlossenen Kartenkacheln durch die im Inneren der Ellipse liegenden $\vec{P}_K$ gegeben:

$$|\vec{P}_K - \vec{P}_1| + |\vec{P}_K - \vec{P}_2| < 2A$$

Dabei ist A die große Halbachse der Ellipse und

$$|\vec{P}_K - \vec{P}_1|^2 = (x_K - x_1)^2 + (y - y_1)^2.$$

Der hier benutzte Parameter für die Kennzeichnung der Ellipse ist das Verhältnis f von großer Halbachse A und kleiner Halbachse $B = \sqrt{A^2 - E^2}$. Der Zusammenhang zur Exzentrizität und der Fläche $F = \pi \cdot A \cdot B$ der Ellipse ist

$$E = \sqrt{\frac{F}{\pi}(f - \frac{1}{f})}.$$

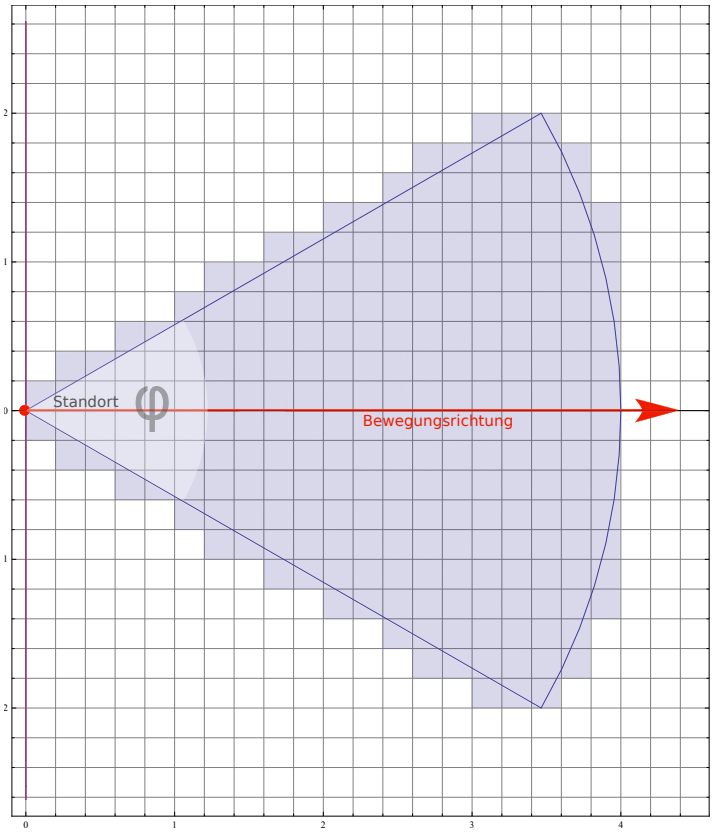
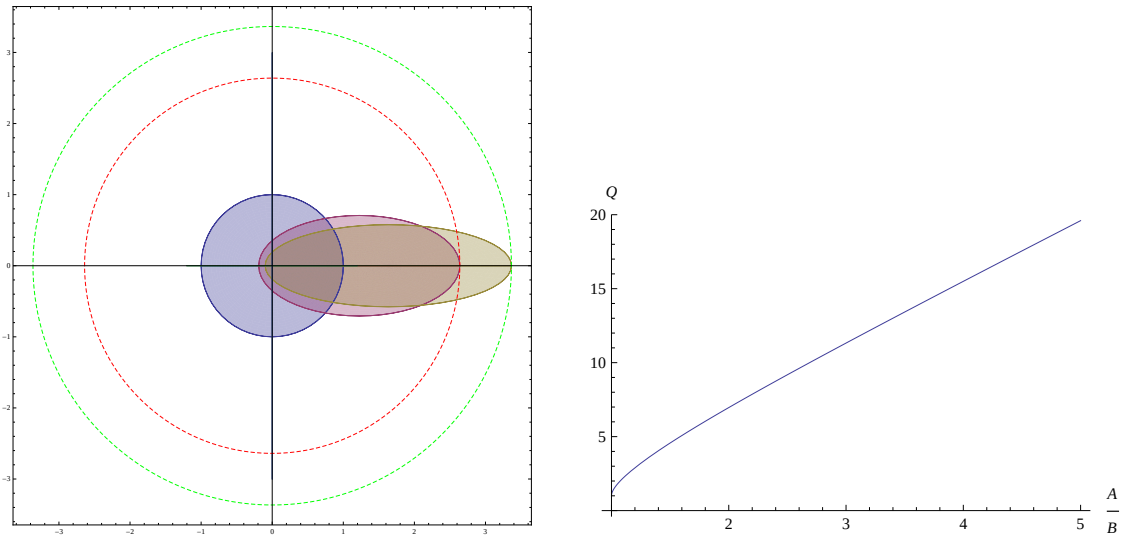


Abbildung 3.5: Geometrisches Cachingverfahren: kreissektorförmiges Caching

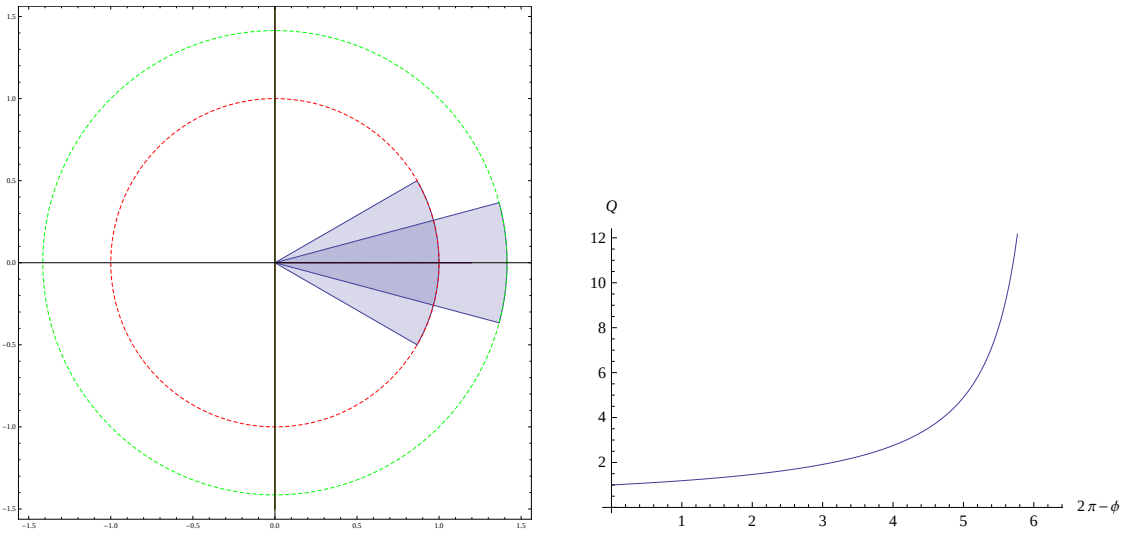
Kreissektorförmiges Preaching Das kreissektorförmige Caching ist wie das elliptische Caching ebenfalls positions- und bewegungsabhängig, wobei hier anstatt einer Ellipse ein Kreissektor mit dem Standort des Clients als Kreismittelpunkt gebildet wird. Dabei bildet die Gerade in Bewegungsrichtung die Winkelhalbierende des Mittelpunktswinkels φ (siehe Abbildung 3.5 bzw. Abbildung 7.1 im Anhang). Abhängig vom Mittelpunktswinkel φ im Bogenmaß und der Gesamtfläche A des Kreissektors beträgt der Radius R des Kreissektors

$$R = \sqrt{\frac{2A}{\varphi}}.$$



(a) (b)

Abbildung 3.6: Motivation für elliptisches Precaching



(a) (b)

Abbildung 3.7: Motivation für kreissektorförmiges Precaching

Je zielgenauer ein geometrisch eingeschränktes Precaching (elliptisch oder kreissektorförmig) die reale Bahn (Trace) des Clients trifft, um so bedeutender ist der Gewinn gegenüber dem kreisförmigen Precaching ohne Richtungspräferenz, die die Bahn ebenso erfasst. Der Gewinn manifestiert sich grob im Verhältnis Q der Flächen ($\sim$ Anzahl der Kartenkacheln) der überdeckenden Kreise zu den Flächen der Ellipsen bzw. Kreissektoren: Dieses Verhältnis spiegelt das reduzierte Datenvolumen durch die Einbeziehung der Richtungsinformation der Bewegung des Clients wider (siehe Abbildung 3.6 (a) mit $f = \frac{A}{B} = 1, 2, 3$ und 3.7 (a) mit $\varphi = \frac{\pi}{3}, \frac{\pi}{6}$. Die Abbildungen 3.6 (b) bzw. 3.7 (b) geben die Abhängigkeiten des Verhältnisses der Flächen der gestrichelt gezeichneten Kreise (Flächen beim kreisförmigen Precaching) zu den elliptischen bzw. kreissektorförmigen Flächen als Funktion des Ellipsenparameters $f = \frac{A}{B}$ bzw. des Mittelpunktswinkels φ an. Große Werte von Q bedeuten potentiell große Einsparungen an über das Netzwerk zu transportierendem Datenvolumen.

(siehe Abbildung 3.6 und 3.7: Die weißen Bereiche der rot, bzw. grün gezeichneten Kreise spiegeln das gegenüber kreisförmigem Precaching eingesparte Datenvolumen wider.)

Nicht-geometrische Verfahren

Nicht-geometrische Caching-Verfahren werden in dieser Arbeit in Kombination mit geometrischen Cachingverfahren, speziell mit dem kreissektorförmigen Precachingverfahren, benutzt. Das Ziel ist es, nach Vorgabe einer groben Einengung des Kartengebiets, das in den lokalen Cache des Clients aufgenommen werden soll, Gebiete innerhalb dieser Vorgabe zu wichten und entsprechend ihrer Wichtung zu bevorzugen. Dadurch entsteht eine Verformung des Kreissektors, wobei nun Kartenmaterial wichtiger Gebiete zur Ergebnismenge des Precachingverfahrens hinzukommen und Kartenmaterial unwichtiger Gebiete dafür verworfen wird.

Precaching, basierend auf Potentialen Dieses Precachingverfahren basiert auf der Annahme, dass von Clients in der Vergangenheit oft benutzte Kartenkacheln auch in der Zukunft häufig benötigt werden. Daher wird jeder Kachel ein Gewicht zugeordnet. Die Menge der Ergebniskacheln einer Anfrage wird iterativ, ausgehend von der Standortkachel K_s des Clients, als erstes Element der Ergebnismenge erzeugt. Es wird sukzessiv diejenige Nachbarkachel K_i der bisherigen Ergebnismenge hinzugefügt, die das größte "Potential"

$$\sum_{\substack{\forall j \\ j \neq i}} \frac{w_j}{d_{ij}^b} \tag{3.1}$$

hat. Die benutzen Variablen haben folgende Bedeutung:

- w_j: Gewicht der Kachel K_j
- d_{ij}: Abstand der Kacheln K_i und K_j
- b: konstanter Exponent

Jeder Client übermittelt nach einer Anfrage eine grobe Darstellung des seit der letzten Anfrage zurückgelegten Weges. Bei der serverseitigen Auswertung dieser Informationen wird zunächst das bisherige Gewicht einer vom Client berührten Kachel mit einem Alterungsfaktor

$$h = 2^{-\frac{t}{t_h}}$$

multipliziert und danach um eins erhöht. Dabei ist t_h die Halbwertszeit der "Wichtigkeit" der Gewichtsinformation und t das "Alter" der Information. Auf diese Weise entsteht eine Gewichtsverteilung im Rastermodell der Karte, die, abgesehen von der Alterung von Informationen, einer Potentialverteilung in der Physik ähnelt. Nach diesem Modell gibt der Algorithmus diejenigen Kartenkacheln zurück, die sich entlang der durch den derzeitigen Standort des Clients verlaufenden Feldlinien befinden.

Für das sinnvolle Funktionieren dieses Precachingalgorithmus ist eine hinreichende Dichte von Benutzern in einem Gebiet notwendig, die für die Erstellung einer statistisch aussagekräftigen Wichtung von Kartenkacheln sorgen.

Precaching, basierend auf Ähnlichkeit von Umgebungen Dieses Precachingverfahren basiert auf der Annahme, dass sich Clients in ähnlichen Umgebungen ähnlich verhalten. Zwei Gebiete bzw. Kartenkacheln sind hier als ähnlich definiert, gdw. sie POIs derselben Sorte enthalten. Anhand der Weginformationen, die von den Clients auch beim o.g. Precachingverfahren, basierend auf Potentialen, übermittelt werden, wird serverseitig ein gewichteter und gerichteter Graph $G = (V, E)$ mit der Bewertungsmatrix $B \in \mathbb{R}^{|V|\times|V|}$ erstellt, dessen Knoten V Sorten von POIs und für dessen Kanten $E \in V \times V$ gilt. Das Matrixelement B_{si} ist die relative Häufigkeit der Bewegung von Clients zwischen POIs der Sorte s nach Sorte i. Bei einer Anfrage eines Clients an den MOSM-Server werden nun zunächst die Sorten von POIs bestimmt, die sich auf der Standortkachel K_s des Clients befinden. Im Folgenden werden ausgehend von der Standortkachel K_s, dem ersten Element der Ergebnismenge, sukzessiv diejenigen Nachbarkacheln K_i der Ergebnismenge zur Ergebnismenge hinzugefügt, für die analog zum obigen Ausdruck 3.1

$$\sum_{\substack{\forall j \\ j\neq i}} \frac{\hat{B}_{sj}}{d^b_{ij}}, \quad \hat{B}_{sj} = \sum_{m\in POI_s} \sum_{n\in POI_j} B_{mn}$$

am größten ist. Hierbei ist POI_s die Indexmenge der POIs im von K_s dargestellten Gebiet und POI_j die Indexmenge der POIs im von K_j dargestellten Gebiet.

Bei dieser Form des Precachings ist nicht die hinreichende Dichte von Nutzern in einem Gebiet, sondern von Nutzern in ähnlichen Umgebungen notwendig, um einen statistisch aussagekräftigen Graphen G zu konstruieren.

3.4.4 Resultierende Architektur des Servers

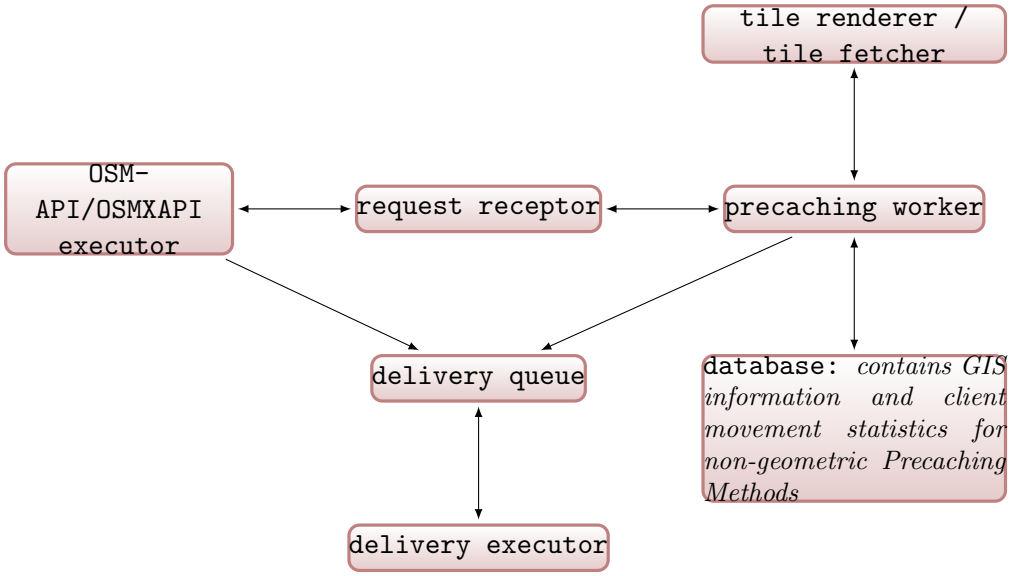


Abbildung 3.8: MOSM-Server-Architektur

Die in Abbildung 3.8 dargestellte Architektur enthält die Komponenten, die zur Umsetzung der o.g. Anforderungen und der gewünschten Anwendungslogik benötigt werden und folgende Aufgaben übernehmen:

- **request receptor:**
  - Empfangen und Weiterleiten von Service Requests
- **precaching worker:**
  - Umsetzten des Precaching-Algorithmus und Liefern einer Ergebnismenge, bestehend aus Metainformationen zu Kartenkacheln
  - Aufrufen des "Tile Renderes" oder des "Tile Fetchers", um entsprechende Bilddateien zu erhalten
  - Übergeben des Ergebnisses an die "Delivery Queue"
- **tile renderer/tile fetcher:**
  - Rendern/Holen von Kartenkachel-Bilddateien
- **database:**
  - Bereitstellen von GIS-Informationen und anonymen statistischen Bewegungsinformationen von Clients für die Ausführung nicht-geometrischer Precachingverfahren.
- **OSM-API/OSM Extended API (OSMXAPI) executor:**
  - Ausführen von lesenden und schreibenden Methoden auf der OSM-Datenbank

- **delivery queue:**
  - Bereithalten von fertigen Ergebnissen für Clients
- **delivery executor:**
  - Ausliefern von Ergebnisankündigungen und Ergebnisdaten sowie Behandeln von Fehlern, verursacht durch die plötzlich abgebrochene Verbindung zu einem Client (siehe 3.4.2)

4 Verwendete Softwarekomponenten und Dienste

4.1 PostgreSQL (v8.3)

PostgreSQL ist ein objektrelationales Datenbanksystem und steht unter BSD-Lizenz. Ab Version 8.0 läuft PostgreSQL nativ auf Windows und gängigen Unix-Derivaten (Linux, AIX, BSD, HP-UX, SGI IRIX, Mac OS X, Solaris, Tru64). Die SQL-Implementation der Datenbank ist streng konform zum ANSI-SQL-2008-Standard. Es existieren Programmierschnittstellen für eine Vielzahl von Programmiersprachen (z.B. C, C++, Java, C#, Python, Ruby, PHP), wodurch PostgreSQL sehr leicht in vielen Umgebungen einsetzbar ist. PostgreSQL ist erweiterbar durch User Defined Functions, die ebenfalls eine Vielzahl von Programmiersprachen (z.B. Java, Perl, Python, Ruby, Tcl, C/C++) und der Datenbank-eigenen Sprache PL/pgSQL erlauben. Inzwischen existieren eine Reihe von Erweiterungspaketen für PostgreSQL in Form von User Defined Functions. Dazu gehören u.a. PostGIS und DbLink. DbLink ist eine Sammlung von Funktionen, die Abfragen über Datenbankgrenzen hinweg ermöglichen. PostGIS erweitert PostgreSQL um die Funktionalität eines geografischen Informationssystems. Wesentliche Funktionen und Eigenschaften von PostGIS werden im folgenden Abschnitt beschrieben.

4.2 PostGIS (v1.4.0)

PostGIS ist eine Erweiterung des PostgreSQL-Datenbankservers und verleiht diesem die Fähigkeit, als Backend-Server für geografische Informationssysteme zu fungieren. Es ist in der Lage, Repräsentationen räumlicher Objekte zu erstellen, zu aktualisieren, zu löschen und auf sie bezogene Abfragen durchzuführen. PostGIS ist konform zur OpenGIS *Simple Features Specification for SQL* und diesbezüglich Open Geospatial Consortium (OGC)-zertifiziert.

4.2.1 Darstellung räumlicher Objekte

Objekte sind in PostGIS konform zur OpenGIS-Spezifikation im Well Known Text (WKT)-Format[1] oder im Well Known Binary (WKB)-Format[2] darstellbar. Diese Formate unterscheiden sich zwar in ihrer Darstellungsform, nicht jedoch in ihrer Semantik. Der Lesbarkeit halber wird in den folgenden Erklärungen und Beispielen das WKT-Format verwendet. PostGIS unterstützt ebenso ein erweitertes WKT- und WKB-Format (Extended Well Known Text (EWKT) und Extended Well Known Binary (EWKB)). Diese Formate unterstützen, im Gegensatz zum OpenGIS-Standard, nicht nur zweidimensionale, sondern z.B. auch dreidimensionale Objekte.

[1] WKT-Format: textbasiertes Format zur Darstellung geometrischer Objekte
[2] WKB-Format: binäres Format zur Darstellung geometrischer Objekte

Basis-Geometrietypen nach OpenGIS

Alle Geometrietypen leiten sich nach der OpenGIS-Spezifikation vom Basistyp GEOMETRY ab. Die Hierarchie der abgeleiteten Klassen ist in Abbildung 4.1 dargestellt, wobei abstrakte Klassen rot und instanziierbare grün dargestellt sind.

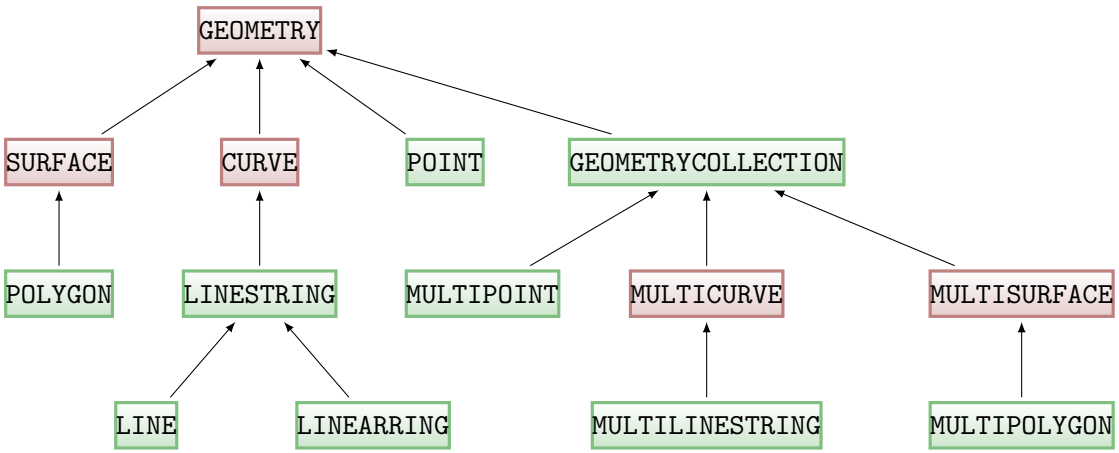


Abbildung 4.1: OpenGIS Geometrietypen-Klassenhierarchie

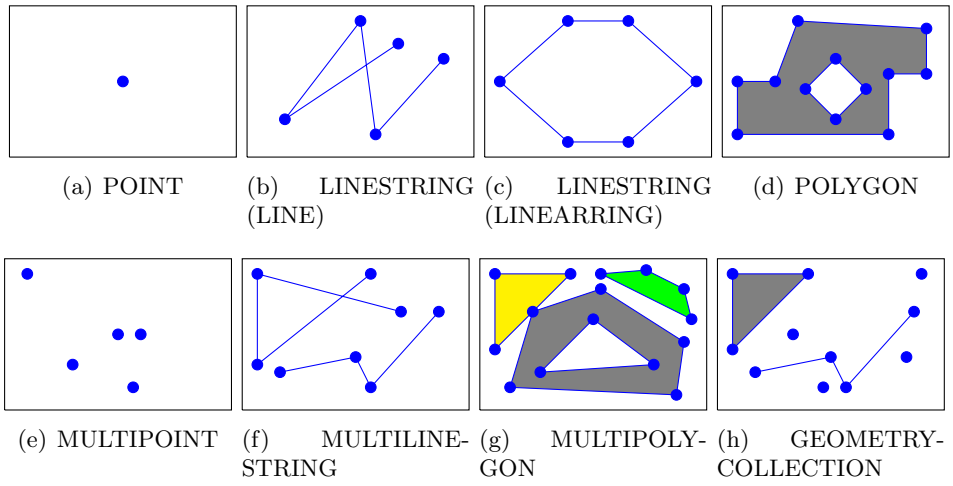
PostGIS kennt drei instanziierbare Geometrietypen, aus denen sämtliche geometrischen Objekte in PostGIS zusammengesetzt sind (siehe Abbildung 4.2 (a) bis (d)):

1. POINT
   - Ein POINT ist eine nulldimensionale, duch eine x- und eine y-Koordinate definierte Geometrie.
2. LINESTRING
   - Ein LINESTRING ist eine eindimensionale, durch Aneinanderreihung von Punkten definierte Geometrie.
   - Eine LINE ist ein LINESTRING, der aus genau zwei Punkten besteht.
   - Ein LINEARRING ist ein LINESTRING, dessen Anfangspunkt auch sein Endpunkt ist und der nicht zweimal durch denselben Punkt verläuft.
3. POLYGON
   - Ein POLYGON ist eine zweidimensionale, durch LINEARRING-Objekte als innere und äußere Grenze definierte Geometrie.
   - Ein POLYGON kann von anderen LINEARRING-Objekten tangiert, nicht jedoch geschnitten werden.
   - Löcher in POLYGON-Objekten, können durch innere LINEARRING-Objekte definiert werden, wobei der Außenbereich immer zusammenhängend bleibt.

Eine Menge von Objekten eines jeweiligen Typs lässt sich mit den folgenden Typen (siehe Abbildung 4.2 (e)-(g)) zu einem Objekt zusammenfassen:

1. MULTIPOINT
   - Ein MULTIPOINT ist eine nulldimensionale, aus einer Menge von POINT-Objekten bestehende Geometrie.
2. MULTILINESTRING
   - Ein MULTILINESTRING ist ein geometrisches Objekt, bestehend aus LINESTRING-Objekten.
3. MULTIPOLYGON
   - Ein MULTIPOLYGON ist ein zweidimensionales geometrisches Objekt, bestehend aus POLYGON-Objekten und enthält weder sich überschneidenden POLYGON-Objekte noch geschnittene LINESTRING-Objekte.

Eine Menge von Objekten unterschiedlichen Typs lässt sich mit dem Typ GEOMETRY-COLLECTION zu einem Objekt zusammenfassen (siehe Abbildung 4.2 (h)).



(a) POINT (b) LINESTRING (LINE) (c) LINESTRING (LINEARRING) (d) POLYGON

(e) MULTIPOINT (f) MULTILINE-STRING (g) MULTIPOLY-GON (h) GEOMETRY-COLLECTION

Abbildung 4.2: Beispiele von Geometrietypen

4.2.2 Erstellen einer PostGIS Datenbank

Das Listing 4.1 demonstriert die Erstellung einer initialen PostGIS-Datenbank.

```
createdb <dbname>
createlang plpgsql <dbname>
psql -f <path to lwpostgis.sql> <dbname>
psql -f <path to spatial_ref_sys.sql> <dbname>
```

Listing 4.1: Shell Befehlszeile: Erstellen einer PostGIS Datenbank

Während Tabellen ohne Geometriespalte in PostgreSQL mit dem Data Definition Language (DDL)-Ausdruck `CREATE TABLE <tablename> (...)` erstellt werden, müssen Spalten eines PostGIS-GEOMETRY-Typs einzeln mit der PostGIS-Methode

```
AddGeometryColumn(varchar table_name, varchar column_name, integer srid,
    varchar type, integer dimension)
```

erzeugt, d.h., an eine existierende Tabelle angefügt werden. Dabei bedeutet

- **table_name:** Name der Tabelle, die eine Geometriespalte erhalten soll
- **column_name:** Name der Geometriespalte
- **srid:** Spatial Reference System Identifier (SRID)
  - Die SRID identifiziert die Art der Projektion, d.h. das Koordinatensystem der in einer Tabelle gespeicherten Objekte. PostGIS verwendet als Default (SRID=-1) die Gauß-Krüger-Projektion, Openstreetmap verwendet in der für Mapnik lesbaren Form die Mercator-Projektion (SRID=900913). SRID verweist auf den Hauptschlüssel der Tabelle "spatial_ref_sys", in der mathematische Spezifikationen zu verschiedenen Projektionsverfahren (darunter Gauß-Krüger und Mercator) abgelegt sind. Diese Spezifikationen werden z.B. bei der Transformation von Koordinaten zwischen verschiedenen Koordinatensystemen (≡ Projektionsverfahren) benötigt.
- **type:** einer der oben beschriebenen Geometrietypen (z.B. 'POINT', 'POLYGON', 'MULTIPOLYGON')
- **dimension:** Dimension des für die Speicherung von Geometrien zugrundegelegten Koordinatensystems (bei Verwendung des OpenGIS-Standards ist dimension=2, bei EWKT bzw. EWKB dimension=2 oder dimension=3)

Die so an PostGIS übergebenen Parameter werden in der Tabelle "geometry_columns" abgespeichert, um später als Metainformation für PostGIS-Methoden verfügbar zu sein.

4.2.3 Einige PostGIS-Methoden

In diesem Abschnitt wird die Funktionalität einiger PostGIS-Methoden[3] kurz vorgestellt. Für die gesamte Spezifikation aller PostGIS-Methoden, siehe [pos].

- **ST_GeometryFromText(text, SRID):** erstellt ein Geometrie-Objekt aus der im 'text'-Parameter gegebenen WKT-Darstellung einer Geometrie
  (z.B. 'POINT(13.2 53.7)', 'LINESTRING(13.2 53.7, 13.2 53.8, 13.3 53.8, 13.3 53.7, 13.2

[3] PostGIS enthält insgesamt über 650 Funktionen

53.7)' oder 'POLYGON((13.2 53.7, 13.2 53.8, 13.3 53.8, 13.3 53.7, 13.2 53.7))'. Der Parameter 'SRID' kennzeichnet den Spatial Reference System Identifier der spezifizierten Geometrie.

- **ST_AsText(geom):** zeigt ein Geometrie-Object im WKT-Format an (ohne SRID).
- **ST_AsBinary(geom):** zeigt ein Geometrie-Object im WKB-Format an (ohne SRID).
- **ST_Transform(geom,SRID):** transformiert eine Geometrie-Objekt eines SRID-Typs in ein Geometrie-Objekt des durch den SRID-Parameter spezifizierten SRID-Typs.
- **ST_Distance(geomA, geomB):** berechnet die Entfernung zweier Geometrien voneinander. Beide Geometrien müssen vom gleichen SRID-Typ sein.
- **ST_Intersection(geomA, geomB):** berechnet die Schnittgeometrie zweier Geometrien
- **ST_IsEmpty(geom):** überprüft, ob eine Geometrie leer ist (z.B.

  ```
  SELECT ST_IsEmpty(ST_Intersection(geomA, geomB))
  ```

  gibt *True* zurück, falls geomA und geomB sich nicht schneiden)
- **ST_Contains(geomA,geomB):** gibt genau dann *True* zurück, wenn kein Punkt von geomB außerhalb von geomA liegt, aber wenigstens einer in geomA liegt, ansonsten 'False'. Beide Geometrien müssen von demselben SRID-Typ sein.
- **ST_Within(geomA,geomB):** gibt *True* zurück, wenn geomA vollständig in geomB liegt, ansonsten 'False'.

4.3 Openstreetmap-API (v0.6)

Openstreetmap stellt eine HTTP-basierte API zur Verfügung, die das Lesen, Erstellen, Aktualisieren und Löschen von Elementen der OSM-Datenbank erlaubt. Während lesende Zugriffe von Jedermann ausführbar sind, benötigen schreibende Zugriffe die Authentifizierung eines Nutzers über *HTTP Basic Authentication* oder *OAuth* (siehe 4.3.5).

4.3.1 Openstreetmap-Datenmodell

Das Openstreetmap-Datenmodell verwendet drei Datentypen, um geografische Objekte darzustellen: Knoten (Nodes), Wege (Ways) und Relationen (Relations). Relationen sind dabei die entscheidene Komponente, um die aus PostGIS bekannten und in allen geografischen Informationssystemen üblichen Objekttypen wie Polygone und Multipolygone zu bauen.

4.3.2 Changesets

Sämtliche Veränderungen an der OSM-Datenbank geschehen innerhalb sogenannter *Changesets*. Diese Gruppierungen von sonst einzeln erscheinenden Änderungen in der OSM-Datenbank dienen nachträglich der besseren Erkennbarkeit von zusammenhängenden Änderungen. Changesets können vom Nutzer geöffnet und geschlossen werden, unterliegen dabei allerdings auch

Beschränkungen. So ist ein Changeset maximal 24 Stunden offen, bevor er automatisch geschlossen wird, bzw. maximal eine Stunde, falls innerhalb dieses Zeitraums kein Schreibzugriff erfolgt. Ein Changeset kann maximal 50.000 Schreibzugriffe (d.h. Creates, Updates oder Deletes) enthalten. Jeder weitere Schreibzugriff erfordert dann das Öffnen eines neuen Changesets.

4.3.3 Einige API Calls

Die hier aufgelisteten und beschriebenen API Calls stellen keine vollständige Auflistung der vorhandenen API dar. Es wurden jene Funktionen ausgewählt, die elementare Funktionalität zum Bearbeiten der OSM-Datenbank bereitstellen. Die Openstreetmap-API in der Version 0.6 ist über die URL http://api.openstreetmap.org/ erreichbar. Ergebnisse vom OSM-Server sowie Text Bodies von HTTP-Requests an den Server werden XML-String übergeben, deren Wurzelknoten immer ein <*osm*>-Element darstellt.

Im Fall eines Fehlers bei der Benutzung der Openstreetmap-API werden entsprechende HTTP-Status-Codes zurückgeliefert. So führt z.B. ein falsch formatierter XML-String innerhalb eines Requests zum HTTP-Status-Code 400 (Bad Request), das Verwenden der falschen HTTP-Methode (z.B. POST statt PUT) zum Status-Code 405 (Method not Allowed) und das Referenzieren eines Changesets, das bereits geschlossen ist, zum Status-Code 409 (Conflict). Im Folgenden werden Fehlercodes nicht betrachtet. Für die vollständige Dokumentation der Openstreetmap-API in der Version 0.6, siehe [osm].

Retrieve Map Data: Über ein HTTP-GET an

/api/0.6/map?bbox=left,bottom,right,top

können Kartendaten im XML-Format für einen durch die Bounding Box (bbox) spezifizierten Bereich abgefragt werden. Für die Bounding Box gilt die Beschränkung

$$\Delta lon \cdot \Delta lat \leq 0.25 \cdot \left(\frac{\pi}{180}\right)^2,$$

wobei Δlon und Δlat die Kantenlängen der im Request spezifizierten Bounding Box auf der Einheitskugel im Gradmaß sind. Als Ergebnis wird eine XML-Datei zurückgegeben, welche alle Knoten, Wege und Relationen innerhalb der Bounding Box enthält. Im Fall, dass eine Relation geografische Objekte enthält, die nicht innerhalb der spezifizierten Bounding Box liegen, sind diese Objekte trotz ihrer Referenzierung durch die Relation nicht im Ergebnis der Anfrage enthalten. Sie müssen getrennt über ihre ID abgefragt werden.

Create Changeset: Um Änderungen an der OSM-Datenbank durchführen zu können, muss zunächst ein Changeset erstellt werden. Dies wird durch ein HTTP-PUT an

/api/0.6/changeset/create

realisiert. Der Body des HTTP-PUT enthält einen XML-String mit einem <changeset>-Element als Kind-Element des <osm>-Elements (siehe Listing 4.2).

```
<osm>
  <changeset>
    <tag k="comment" v="setting some POIs in Adlershof"/>
    ...
  </changeset>
</osm>
```

Listing 4.2: Beispiel für das Erstellen eines Changesets

Bei der erfolgreichen Erstellung eines neuen Changesets liefert der Server die ID des Changesets als ASCII-Zeichenkette zurück.

Read Changeset: Informationen über ein Changeset mit bekannter ID können über ein HTTP-GET an

/api/0.6/changeset/#id

abgerufen werden. Als Ergebnis wird ein XML-String mit einem <changeset>-Element als Kind-Element des <osm>-Elements (siehe Listing 4.3) zurückgegeben.

```
<osm version="0.6" generator="OpenStreetMap server">
  <changeset id="1354" user="cotto" uid="21" created_at="2009-11-05T01:43:29Z"
      closed_at="2009-11-05T03:16:19Z" open="false" min_lon="13.5633271"
     min_lat="52.4801788" max_lon="13.5633271" max_lat="52.4801788">
    <tag k="created_by" v="mosm v0.1"/>
    <tag k="comment" v="editing some POIs"/>
  </changeset>
</osm>
```

Listing 4.3: Ergebnis-XML-String für lesenden Zugriff auf ein Changeset

Close Changeset: Neben dem automatischen Schließen eines Changesets nach Eintreten einer der o.g. Bedingungen kann ein Changeset über ein HTTP-PUT an

/api/0.6/changeset/#id/close

auch manuell geschlossen werden. Nach dem erfolgreichen Schließen des Changesets wird bis auf den Status-Code 200 (OK) kein Rückgabewert geliefert.

Create Element: Über ein HTTP-PUT an

/api/0.6/[node|way|relation]/create

kann ein Element, d.h., ein Knoten (Node), ein Weg (Way) oder eine Relation erstellt werden. Das Kind-Element des <osm>-Elements des XML-Strings im Body des HTTP-Requests ist je nach Typ des zu erstellenden Elements ein <node>-, <way>- oder <relation>-Element (siehe Listings 4.4,4.5,4.6).

```
<node changeset="1355" lat="52.4295452" lon="13.5303601">
  <tag k="shop" v="Copyshop"/>
  <tag k="name" v="CSV Copy"/>
  <tag k="comment" v="Copyshop im Johann von Neumann-Haus"/>
  ...
</node>
```

Listing 4.4: Beispiel: XML-Repräsentation eines Knoten für schreibenden Zugriff

```
<way changeset="1355">
  <tag k="comment" v="Ein Weg"/>
  <nd ref="10001"/>
  <nd ref="10002"/>
  <nd ref="10003"/>
  ...
</way>
```

Listing 4.5: Beispiel: XML-Repräsentation eines Weges für schreibenden Zugriff

```
<relation changeset="1355">
  <tag k="type" v="multipolygon" />
  <member type="way" id="2345" role="outer" />
  <member type="way" id="2354" role="inner" />
</relation>
```

Listing 4.6: Beispiel: XML-Repräsentation einer Relation für schreibenden Zugriff (Erzeugen eines Multipolygons: Die Fläche des Multipoligons ergibt sich aus der Fläche des äußeren (outer) Polygons abzüglich der Fläche des inneren (inner) Polygons.)

Read Element: Über ein HTTP-GET an

/api/0.6/[node|way|relation]/#id

kann ein Element (Knoten, Weg oder Relation) in seiner XML-Darstellung abgerufen werden. Dabei werden auch serverseitig automatisch generierte Attribute des jeweiligen Elements wie Version, Nutzername und ID des Erstellers des Elements, und der Zeitstempel der letzten Veränderung des Elements zurückgegeben.

Update Element: Über ein HTTP-PUT an

/api/0.6/[node|way|relation]/#id

kann ein bereits bestehendes Element aktualisiert werden. Dabei muss die gesamte XML-Repräsentation des aktualisierten Elements übergeben werden und mit der aktuellen Versionsnummer als Attribut des Elements versehen sein. Bei erfolgreicher Aktualisierung wird die inkrementierte Versionsnummer zurückgegeben. Wird eine falsche oder veraltete Versionsnummer übergeben, so führt dies zu einem Fehler (Status Code 400 (Bad Request)).

Delete Element: Über ein HTTP-DELETE an

/api/0.6/[node|way|relation]/#id

kann ein Element gelöscht werden. Wie auch beim Update eines Elements verlangt diese Methode die Übergabe der vollständigen XML-Repräsentation des zu löschenden Elements einschließlich aktueller Versionsnummer.

4.3.4 Erweiterte Openstreetmap-API (OSMXAPI)

Die OSM Extended API (OSMXAPI) stellt eine read-only-API zur Openstreetmap-Datenbank dar. Sie arbeitet auf einer Spiegelung der Openstreetmap-Datenbank, womit Änderungen an der originalen Openstreetmap-Datenbank nicht sofort übernommen werden. Laut Dokumentation der API beträgt der für eine Synchronisation einer Änderung benötigte Zeitraum normalerweise nicht mehr als 10 Minuten. Die OSMXAPI ist unter http://www.informationfreeway.org verfügbar.

Erweiterte Funktionalität der API

Verglichen mit lesenden Methoden der Openstreetmap-API ist die OSMXAPI in der Lage, wesentlich differenziertere Suchanfragen zu stellen. Über einen HTTP-GET an

/api/0.6/node[...] oder /api/0.6/way[...] oder /api/0.6/relation[...]

kann eine gezielte Anfrage über einen oder mehrere Knoten, Wege bzw. Relationen gestellt werden. Der "[...]"-Ausdruck hinter den URLs steht dabei für die im Folgenden beschriebenen Prädikate, die die Suche nach Elementen eingrenzen bzw. definieren:
Es gibt drei verschiedene Sorten von Prädikaten:

1. **Tag Predicates:** Über diese Prädikate lassen sich Elemente anhand ihrer Tags suchen. Es kann sowohl nach dem exakten Wert eines Tags als auch nach dem bloßen Vorhandensein eines bestimmten Typs von Tags gesucht werden. Mit Hilfe des "|"-Operators lassen sich verschiedene Bedingungen oder-verknüpfen.
   - Beispiel: node[amenity=fuel|leisure]
   - Erklärung: Es werden Knoten gesucht, die entweder Tankstellen darstellen oder einen Tag vom Typ "leisure" besitzen.
2. **BBox Predicates:** Über diese Prädikate kann die Suche nach Elementen örtlich eingeschränkt werden. Wie schon bei der Openstreetmap-API wird die Bounding Box über ihre Kanten definiert: [bbox=left,bottom,right,top].
3. **Child Element Predicates:** Um Elemente mit bestimmten Typen von Kind-Elementen zu suchen, sind diese als Prädikate in der folgenden Schreibweise spezifizierbar:

   $$\text{Child Element Predicates} ::= (not)?[tag|nd|way|relation].$$

   Somit ist spezifizierbar, ob die gesuchten Elemente Tags, Knoten, Wege oder Relationen als Kind-Elemente enthalten müssen oder dieses gerade nicht dürfen.

4.3.5 OAuth

OAuth ist ein offenes Protokoll, das die Autorisierung eines fremden Dienstes (Consumer) für den Zugriff auf geschützte Daten (Protected Ressources) eines anderen Web-Dienstes (Service Provider) ermöglicht, ohne dabei Berechtigungsnachweise des Nutzers selbst weiterzugeben (z.B. Nutzername und Passwort, siehe [Mar07]). Die OSM-API v0.6 bietet diese Art der Autorisierung unter Verwendung von OAuth 1.0 an. Nutzername und Passwort werden im Laufe des Autorisationsverfahrens durch ein Access Token und ein Access Token Secret ersetzt, die vom Consumer benutzt werden, um auf geschützte Daten zuzugreifen. Für diese Version von OAuth ist eine Sicherheitslücke bekannt (dreibeiniges OAuth), die durch OAuth 1.0a (siehe [Mar09]) (derzeit nicht von OSM unterstützt) geschlossen wurde.

4.4 Rendering von Karten - Mapnik & Tiles@Home

4.4.1 Mapnik

Mapnik ist eine Werkzeugsammlung zum Rendern von Karten. Es ist in der Lage, verschiedene Datenquellen zu verwenden, darunter u.a. die PostGIS Datenbank in Kombination mit externen Daten zur Darstellung der Küstenlinien in kleinen Zoomstufen. Für die Nutzung von Mapnik mit OpenStreetmap-Daten muss zunächst die PostGIS-Datenbank mit Daten bestückt werden. Dies übernimmt das Werkzeug "osm2pgsql". Openstreetmap stellt in regelmäßigen Abständen Dumps der aktuellen Openstreetmap Datenbank[4] öffentlich bereit. Ein solcher Dump lässt sich, wie in Listing 4.7 demonstriert, einfach in eine leere PostGIS-Datenbank importieren.

```
1 wget http://planet.openstreetmap.org/planet-latest.osm.bz2
2 osm2pgsql -m -d <dbname> planet-latest.osm.bz2
```

Listing 4.7: Shell Befehlszeile: PostGIS Datenimport

Die so bestückte Datenbank wird von Mapnik benötigt, um Karten zu rendern. Das Erscheinungsbild der entstehenden Kartenkacheln ist dabei über ein Style-Sheet konfigurierbar. In diesem wird definiert, wie Objekte zu rendern sind, wobei z.B. Liniendicken, Farben und ihre Deckkraft definiert werden können.

4.4.2 Tiles@Home

Tiles@Home ist ein verteilter Dienst, um Kartenkacheln zu rendern. Ein zentraler Server (tah.openstreetmap.org) ist mit einer großen Anzahl von Clients, z.B. Heimrechnern von Personen, die einen Teil ihrer Rechenleistung dem Tiles@Home-Projekt zur Verfügung stellen, verbunden und verteilt an diese Jobs zum Rendern von Kartenkacheln. Die mit dem Werkzeug *Osmarender* fertiggestellten Kartenkacheln werden zurück zum Server gesendet und können über die Tiles@Home-API mit einem HTTP-GET Request auf

```
http://tah.openstreetmap.org/Tiles/tile/<zoom>/<longitude index>/<latitude
    index>.png
```

[4] http://planet.openstreetmap.org

abgerufen werden. Über ein HTTP-POST an die URL[5]

```
http://tah.openstreetmap.org/Request/create/?x=<longitude index>&y=<latitude
    index>&priority=<[1..3]>
```

können Kartenkacheln neu erstellt werden. Die Kachelindizes beziehen sich dabei auf die Zoomstufe 12. Erstellt werden Kacheln der Zoomstufe 12 und alle Kartenkacheln höherer Zoomstufen, die in ihr enthalten sind.

4.5 OpenLayers (v2.8)

OpenLayers ist eine reine JavaScript-Bibliothek mit dem Zweck, geografische Daten in einem Browserfenster anzuzeigen. Dabei unterstützt OpenLayers die Darstellung von Daten aus verschiedenen Quellen. Dazu gehört die Darstellung von Geodaten von Openstreetmap, Google Maps, Yahoo Maps und Virtual Earth sowie Datenquellen, die konform zu von OpenGIS standardisierten Datenformaten wie Web Feature Service (WFS) und Web Map Service (WMS) sind. OpenLayers ist in der Lage, Informationen in thematischen Schichten (Layers) anzuordnen. Während z.B. Layer 1 Openstreetmap-Karten enthält, können darüber liegende Schichten POIs (siehe Beispiel in Abbildung 4.3) oder andere GIS-Objekte anzeigen (siehe 4.2.1).

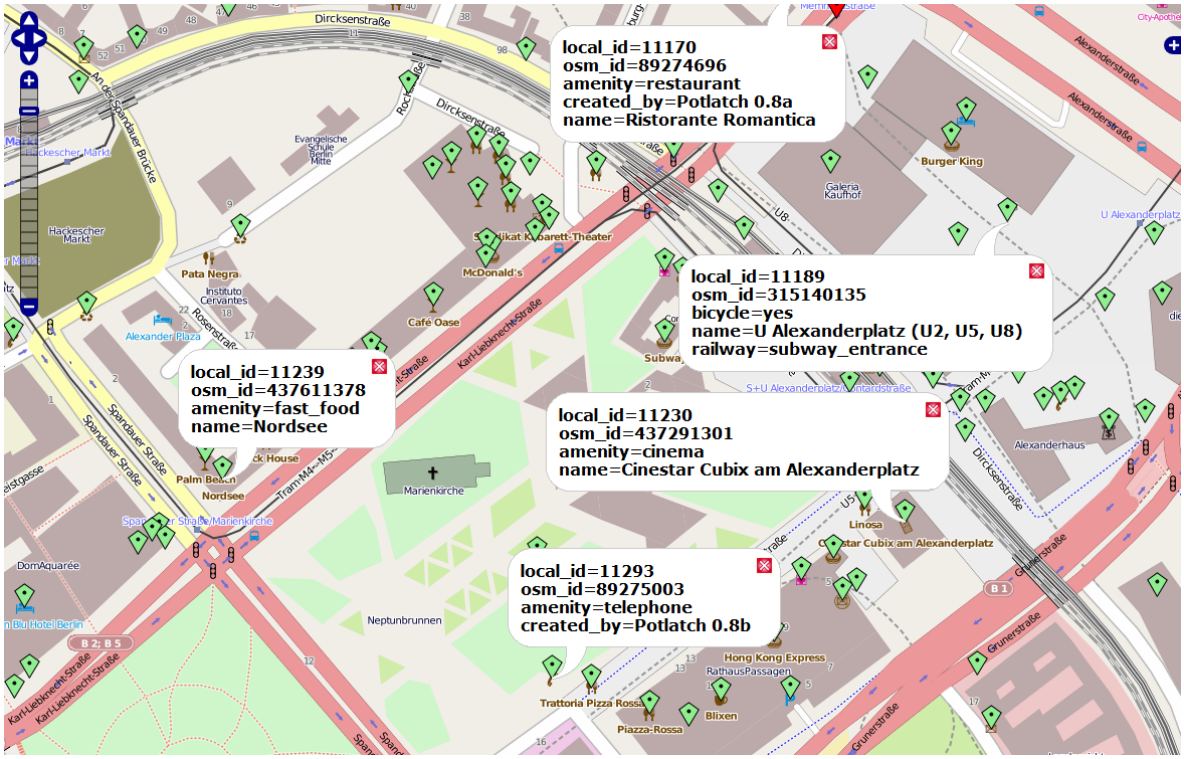


Abbildung 4.3: POIs als thematische Schicht in OpenLayers

[5] d.h. ein HTTP-POST mit dem Content-Type "application/x-www-form-urlencoded" und den Argumenten x=<longitude index>, y=<latitude index> und priority=[1..3], wobei 1 die höchste und 3 die niedrigste Ausführungspriorität darstellt

5 Implementierung

5.1 Implementierung des Servers und Simulators zur Evaluation der Precachingalgorithmen

5.1.1 Server

Funktionalität

Die in Python vorgenommene Implementierung umfasst nicht die gesamte für ein Produktivsystem notwendige Funktionalität, da nur Teile davon für die Auswertung der entwickelten Precachingstrategien relevant sind. Die implementierten Teile sind:

- das Bearbeiten von Map-Requests
- das Verwalten von statistischen, durch Clients erhobene Daten zur Ausführung von nicht-geometrischen Precachingverfahren
- die Realisierung der Alterung[1] von statistischen Daten
- die Ausführung von Precachingberechnungen nach den beschriebenen geometrischen und nicht-geometrischen Verfahren

Die Implementierung des Servers gliedert sich in einen Serverprozess und mehrere Render-Prozesse auf. Letztgenannte rendern mit Hilfe der Mapnik-Bibliothek Kartenkacheln der Größe 256x256 Pixel und speichern diese im PNG-Format. Die Render-Prozesse lassen sich im Netzwerk beliebig verteilen, wodurch eine Rechenlastverteilung auf mehrere Prozessoren bzw. Rechner möglich ist. Alternativ währe ein Herunterladen fertiger Kartenkacheln vom OSM-Tileserver oder von Tiles@Home möglich. An der Datenbank vorgenommene Änderungen (hier nicht implementiert) werden seitens des OSM-Tileservers im Schnitt allerdings erst nach einigen Stunden übernommen. Somit ist das Erstellen von Kartenkacheln direkt durch den MOSM-Server mit eigener aktueller Datenbank sinnvoll, um durch mobile Nutzer vorgenommene Änderungen für diese schneller zu realisieren. Der MOSM-Server-Prozess selbst instanziiert noch einmal vier Thread-Klassen:

- **RequestReceptor**: Diese Thread-Instanz wartet auf einem zuvor konfigurierten TCP-Port auf eingehende Verbindungen, empfängt eingehende Pakete und reiht enthaltene Requests je nach Typ des Pakets und je nach ID des sendenden Clients in die Bearbeitungsqueue ein (s.u. “Scheduler”). Der RequestReceptor reagiert ebenfalls auf das Zusammenbrechen von Verbindungen, indem er das Senden von zu dieser Verbindung gehörenden Paketen in der DeliveryQueue (s.u “WorkerThread” und “ResultDeliverer”)

[1] Die Aussage, die mit Hilfe von statistischen Daten über ein geografisches Gebiet getroffen werden kann, wird mit dem Alter dieser Daten ungenauer. Deshalb gehen neuere Daten mit einem höheren Gewicht in eine Aussage ein als ältere. Dies wird mit einer Halbwertzeit von statistischen Daten realisiert (siehe 3.4.3).

bis zu einem erneuten Verbinden des entsprechenden Clients zurückstellt. Im Falle eines akkut unterbrochenen Sendevorgangs seitens der DeliveryQueue, übernimmt diese die Fehlerbehandlung.

- **Scheduler**: Diese Thread-Instanz verteilt die Bearbeitung der in der Queue liegenden Requests ihrer Priorität nach auf eine Anzahl von Worker-Threads. Eine Scheduler-Instanz ist notwendig, um Fairness zwischen mehreren gleichzeitig anfragenden Clients zu garantieren. Abhängig davon, wie viele Kartenkacheln für eine Anfrage mit Mapnik neu erstellt werden müssen, ist eine Anfrage nach Kartenkacheln mehr oder weniger rechenintensiv und somit zeitintensiv. Entsprechend dieser für die Bearbeitung einer Anfrage benötigten Rechenzeit werden für jeden Client für die nachfolgenden Anfragen Prioritäten vergeben, nach denen der Scheduler die Bearbeitung einleitet. Im Vorausblick auf einen Ausbau der Implementation ist eine Scheduler-Komponente auch für die Bearbeitung von z.B. Suchanfragen seitens eines Clients sinnvoll, da deren Bearbeitung sich von Fall zu Fall stark im Rechenaufwand unterscheiden kann. Da die Scheduler-Komponente später nicht evaluiert wird, wird hier diesbezüglich nicht auf weitere Details eingegangen.

- **WorkerThread**: Diese Thread-Instanz bearbeitet alle Arten von Requests, die ihr durch den Scheduler übergeben werden. Je nach Typ des Requests wird der WorkerThread-Instanz ein Funktionszeiger zu der Funktion, die für die Behandlung des jeweiligen Request-Typs verantwortlich ist, übergeben. Zu Request-Typen zählen hier auch Pakete, die nur der Erstellung von der serverseitigen Statistik dienen und kein Ergebnispaket verlangen. Sämtliche Ergebnisdaten werden nach ihrer Erstellung in die DeliveryQueue gestellt, aus der diese nach dem First In - First Out (FIFO)-Prinzip gesendet werden.

- **ResultDeliverer**: Diese Thread-Instanz arbeitet Einträge der DeliveryQueue ab und versendet Ergebnispakete zu den dazugehörigen Clients, sobald sie mit dem Server verbunden sind. Fehlgeschlagene Sendeversuche aufgrund einer zusammengebrochenen Verbindung zum Client werden abgefangen und die betroffenen Pakete bis zu einer Neuverbindung des Clients als nicht sendebereit markiert. Auf diese Weise markierte Pakete werden in der DeliveryQueue übersprungen und neu eingereiht.

Datenbankdesign und Funktionalität - Implementierung der Precachingalgorithmen

Die Precachingfunktionalität des Servers ist vollständig in seinem PostgreSQL-Datenbank-Backend in der datenbankeigenen Sprache *PL/pgSQL* implementiert. Die Implementation rein geometrischer Precachingverfahren wäre auch unabhängig von der Datenbank möglich gewesen. Nicht-geometrische Verfahren benötigen allerdings Zugriff auf die PostGIS-Datenbank und weitere eigene Tabellen zur Speicherung statistischer Daten, weshalb die Precaching-Verfahren hier in PL/pgSQL implementiert sind. Für das geometrische Precaching werden die Geometrieparameter in den folgenden Spalten der Tabelle “persistent_variables” festgelegt:

- caching_shape: gibt an, ob elliptisches oder kreissektorförmiges Caching verwendet werden soll (1=elliptisch, 2=kreissektorförmig).

- flattening (f): gibt das Verhältnis von großer Halbachse A zu kleiner Halbachse B für das elliptische Caching an

- sector_angle (φ): gibt den Mittelpunktswinkel für das kreissektorförmige Caching im Gradmaß an

Der Aufruf der User-Defined Function (`SELECT * FROM get_tiles_ps(...)`) liefert dann entsprechend den getroffenen Festlegungen die Kachelindizes der Kacheln, die zur Ergebnismenge des Precachingverfahrens gehören.

Listing 5.1 enthält die Signatur der Funktion "get_tiles_ps".

```
1 get_tiles_ps(IN lon double precision, IN lat double precision, IN lon_speed
    double precision, IN lat_speed double precision, IN res_cnt integer, IN
    lon_range integer, IN lat_range integer, IN zoom integer, OUT lon_index
    integer, OUT lat_index integer) RETURNS SETOF RECORD
```

Listing 5.1: get_tiles_ps Signatur

Dabei spezifizieren "lon" und "lat" die aktuelle geografische Position des Clients, "lon_speed" und "lat_speed" den aktuellen Geschwindigkeitsvektor des Clients[2], "res_cnt" die Anzahl der zurückzuliefernden Kacheln (Mächtigkeit der Ergebnismenge: legt die Flächengröße fest, für die geografische Daten in den Cache des Clients geladen werden) und "zoom" die gewünschte Zoomstufe im Intervall [1..18] der Kartenansicht. "lon_range" und "lat_range" bekommen erst bei nicht-geometrischen Precachingverfahren eine Bedeutung und sollten für geometrische Verfahren auf 0 gesetzt werden.

Die Bestimmung der Ergebnismenge E verläuft iterativ. Sei M die Menge aller Kartenkacheln einer Zoomstufe und E_i die Ergebnismenge nach der i-ten Iteration. Ausgehend von der Standortkachel als initiale Ergebnismenge E_0 wird bei jeder Iteration i diejenige zu einer aus E benachbarten[3] Kachel $K \in M \backslash E_{i-1}$ mit dem zweidimensionalen Index $\vec{P}_K$ zur Ergebnismenge hinzugefügt, für die folgende Bedingung zutrifft:

- **bei elliptischem Precaching:**

  $$|\vec{P}_K - \vec{P}_1| + |\vec{P_K} - \vec{P}_2| \text{ ist minimal,}$$

  wobei $\vec{P}_1$ und $\vec{P}_2$ die Brennpunkte der Ellipse sind ($\vec{P}_1$ ist dabei der zweidimensionale Index der Standortkachel des Clients) (siehe 3.4.3)

- **bei kreissektorförmigem Precaching:**

  $$|\vec{P}_K - \vec{P}_1| \text{ ist minimal und } |\sphericalangle(\vec{P}_K - \vec{P}_1, \vec{e})| \leq \frac{\varphi}{2},$$

  wobei $\vec{P}_1$ der zweidimensionale Kachelindex der Standortkachel des Clients und $\vec{e}$ der Einheitsvektor in Richtung des Geschwindigkeitsvektors ist
  $\left(\vec{e} = \frac{1}{\sqrt{lon\_speed^2 + lat\_speed^2}} \begin{pmatrix} lon\_speed \\ lat\_speed \end{pmatrix}\right)$ (siehe 3.4.3)

Kreisförmiges Precaching lässt sich erreichen, indem entweder "caching_shape" auf kreissektorförmiges Caching und "sector_angle" auf einen Winkel von 360° gesetzt wird oder "caching_shape" auf elliptisches Caching und "flattening" auf 1 gesetzt wird. Auf diese Weise liefert die Funktion "get_tiles_ps" eine Menge von kreisförmig um den aktuellen Standort angeordneten Kartenkacheln.

[2] Da sich der Client tatsächlich auf krummlinigen Koordinaten bewegt, ist die Angabe eines Geschwindigkeitsvektors in diesen krummlinigen Koordinaten für eine geradlinige Bewegung nur eine Näherung für Bewegungen über kurze Entfernungen.

[3] Zwei Kacheln sind benachbart, gdw. sie eine gemeinsame Kante besitzen.

Implementierung des Precachings, basierend auf Potentialen: Basierend auf kreissektorförmigem Caching findet hier innerhalb der durch den Mittelpunktswinkel φ gegebenen Grenzen des Kreissektors eine Wichtung von Kacheln statt. Diese Wichtung wird anhand der Frequentierung des durch die Kachel dargestellten Gebiets durch Clients definiert (s.u.). Analog zu geometrischen Verfahren arbeitet auch dieses Verfahren iterativ, wobei bei jeder Iteration i diejenige zu einer aus der Ergebnismenge E benachbarten Kachel $K \in M \backslash E_{i-1}$ an der Position $\vec{P}_K$ zur Ergebnismenge hinzugefügt wird, für die folgende Bedingung zutrifft:

$$p_{x_K,y_K} = \sum_{\substack{x=x_K-r_x \\ x\neq x_K}}^{x_K+r_x} \sum_{\substack{y=y_K-r_y \\ y\neq y_K}}^{y_K+r_y} \frac{w_{x,y}}{|\vec{P} - \vec{P}_k|^b} \text{ ist maximal und } |\sphericalangle(\vec{P}_K - \vec{P}_1, \vec{e})| \leq \frac{\varphi}{2},$$

wobei

- $\vec{P}_K = \begin{pmatrix} x_K \\ y_K \end{pmatrix}; \quad \vec{P} = \begin{pmatrix} x \\ y \end{pmatrix}$
- p_{x_K,y_K} das Potential der Kachel an der Position P_K ist,
- $w_{x,y}$ gleich dem Gewicht der Kachel mit den Indizes (x,y) ist,
- der Exponent b eine Konstante ist (gegeben durch die Spalte "potential_exponent" der Tabelle "persistent_variables" im Datenbank-Backend)
- und $r_x = lon\_range$ und $r_y = lat\_range$ (ad-hoc über das Server-Konfigurations-File konfigurierbar) sind.

Die für geometrische Precachingverfahren nicht benötigten Parameter lon_range und lat_range geben bei nicht-geometrischen Precachingverfahren die Berechnungsgrenzen für das Potential einer Kachel an. Je größer die Parameter r_x und r_y gewählt werden, um so genauer wird das Potential einer Kachel berechnet. Für $r_x = r_y = r$ ist die Laufzeit von der Ordnung r^2.

In Abbildung 5.1 ist das für dieses Precachingverfahren erstellte Datenbankschema abgebildet.

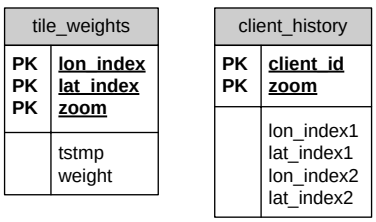


Abbildung 5.1: Datenbankschema für Precaching, basierend auf Potentialen

Der Aufruf der User-Defined Function "inc_weight" (siehe Listing 5.2) erhöht das Gewicht der Kacheln aller Zoomstufen, in denen die durch die Argumente "lon" und "lat" gegebene Position enthalten ist.

```
1 inc_weight(IN lon double precision, IN lat double precision, IN client_id text
    , IN tstmp timestamp) RETURNS VOID
```

Listing 5.2: inc_weight Signatur

Clients senden regelmäßig Trace-Daten zum MOSM-Server. Dieser erhöht daraufhin die Wichtung aller Kacheln, die von dem Trace berührt werden. Einträge in die Tabelle ”client_history“ sollen verhindern, dass einfache Bewegungen, wie das Hin- und Herspringen zwischen zwei benachbarten Kacheln, das Gewicht dieser Kacheln künstlich in die Höhe treiben. Dies kann z.B. zustande kommen, wenn sich ein Client genau auf der Grenze zwischen zwei Kacheln bewegt. Sobald bei der Analyse des Traces für eine Zoomstufe ein Übergang zu einer neuen Kachel (Längengradindex x, Breitengradindex y) festgestellt ist, wird die Tabelle ”client_history“ nach folgendem Schema aktualisiert:

$$\begin{aligned} \text{client\_history.lon\_index2} &:= \text{client\_history.lon\_index1} \\ \text{client\_history.lat\_index2} &:= \text{client\_history.lat\_index1} \end{aligned}$$

und dann

$$\begin{aligned} \text{client\_history.lon\_index1} &:= \text{x} \\ \text{client\_history.lat\_index1} &:= \text{y} \end{aligned}$$

Die Funktion ”inc_weight“ wird nun nur in dem Fall, dass $\begin{pmatrix} client\_history.lon\_index2 \\ client\_history.lat\_index2 \end{pmatrix}$ nicht den Indizes der in der Analyse aktuell berührten Karte entspricht, aufgerufen.

Der mit ”inc_weight“ übergebene Zeitstempel ”tstmp“ markiert den Zeitpunkt des Betretens einer neuen Kachel durch den Client. Da die Wichtung einer Kachel in Abhängigkeit von der Zeit nicht konstant ist[4], sind Erhöhungen eines Kachelgewichts, die in naher Vergangenheit erfolgt sind, auch für die letztendliche Wichtung der Kachel als bedeutender einzustufen als die weiter zurückliegende Inkrementierungen. In der Tabelle ”persistent_variables“ wird zu diesem Zweck eine Halbwertszeit (”tw_half_life“) für die Inkrementierung von Kacheln festgelegt. Ein sinnvoller Wert für diese Größe kann nur plausibel aus Erfahrungswerten gewonnen werden. Aus Mangel an dieser Erfahrung wurde bei den in Kapitel 6 ausgewerteten Messungen dieser Wert auf unendlich viele (bzw. den maximal zulässigen Wert des Datentyps ’interval’) Tage festgelegt, womit die Halbwertszeit für die Inkrementierungen hier keine Rolle mehr spielt.

Implementierung des Precachings, basierend auf der Ähnlichkeit von Umgebungen: Diese Form des Precachings ist eine Variante des Precachings, basierend auf Potentialen. Sie unterscheidet sich in der Art, wie Gewichte von Kacheln zugewiesen werden (siehe 3.4.3). Basierend auf kreissektorförmigem Caching findet auch bei dieser Form des nicht-geometrischen Precachings innerhalb der durch den Mittelpunktswinkel φ gegebenen Grenzen des Kreissektors eine Wichtung von Kacheln statt. War die Wichtung von Kacheln beim Precaching, basierend auf Potentialen, noch unabhängig vom aktuellen Standort eines Clients, so hängt sie jetzt von diesem ab.

Der Aufruf der User-Defined Function `get_tiles_poi(...)` liefert die Kachelindizes der Kacheln, die zur Ergebnismenge des Precachingverfahrens gehören, als Ergebnis zurück.

[4] Im Zuge der zeitlichen Entwicklung eines geografischen Gebiets verlieren oder gewinnen einige Teilgebiete an Bedeutung.

```
get_tiles_poi(IN lon double precision, IN lat double precision, IN lon_speed
    double precision, IN lat_speed double precision, IN res_cnt integer, IN
    lon_range integer, IN lat_range integer, IN client_id text, IN zoom
    integer, OUT lon_index integer, OUT lat_index integer) RETURNS SETOF
    RECORD
```

Listing 5.3: get_tiles_poi Signatur

Die Signatur und die Bedeutung der Parameter sind, mit Ausnahme des Funktionsnamens, identisch zu denen der Funktion ”get_tiles_ps“. In Abbildung 5.2 ist das für dieses Precaching-verfahren erstellte Datenbankschema abgebildet.

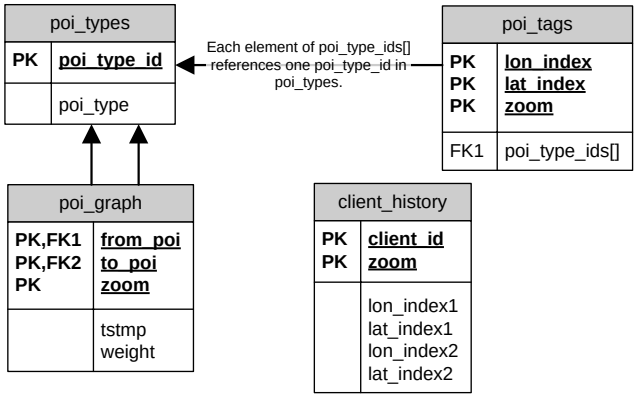


Abbildung 5.2: Datenbankschema für Precaching, basierend auf der Ähnlichkeit von Umgebungen

Die Tabelle ”poi_graph“ enthält Wichtungen von gerichteten Kanten zwischen verschiedenen POI-Typen (z.B. ’restaurant’, ’parking’, ’shop’, ...). Den Ausgangsknoten einer Kante bildet ”from_poi“, den Zielknoten bildet ”to_poi“.

In der Tabelle ”poi_tags“ sind alle Kacheln[5] mit den in ihnen enthaltenen POI-Typen enthalten. Das Array ”poi_type_ids“ umfasst alle Schlüssel zu den in der Tabelle ”poi_types“ gespeicherten Typen von POIs, die in der über ”lon_index“, ”lat_index“ und ”zoom“ spezifizierten Kachel enthalten sind.

Sei die Kachel S die Standortkachel eines Clients mit der Position $\vec{P}_S$ in einer festgelegten Zoomstufe. Die Gewichte der restlichen Kacheln K mit den Positionen $\vec{P}_K$ dieser Zoomstufe ergeben sich nun aus der Summe der Kantenwichtung aller Kanten mit folgenden Eigenschaften:

- Der Ausgangsknoten der Kante stellt einen POI-Typ dar, der im Gebiet der Standortkachel enthalten ist.
- Der Zielknoten der Kante stellt einen POI-Typ dar, der im Gebiet der Kachel $\vec{P}_K$ vorhanden ist.

Mit dieser Festlegung der Gewichte der Kacheln erfolgt die Berechnung der Ergebnismenge E analog zum Precaching, basierend auf Potentialen.

[5] Da sich die Anzahl aller Kacheln aller Zoomstufen auf $\sum_{i=1}^{i=18} 2^{i+1}$ Kacheln beläuft, werden nur wenigstens einmal in einer Berechnung benutzte Kacheln in die Tabelle aufgenommen.

Bei der Analyse der von Clients gesendeten Trace-Daten werden über die User Defined Function ”inc_edge“ (siehe Listing 5.4), analog zu ”inc_weight“, bei Precaching, basierend auf Potentialen, die Gewichte der Kanten des POI-Graphen in ”poi_graph“ inkrementiert.

```
1 inc_edge(IN lon double precision, IN lat double precision, IN client_id text,
    IN tstmp timestamp) RETURNS VOID
```

Listing 5.4: inc_edge Signatur

Es werden nur Kacheln betrachtet, deren dargestellte Gebiete POIs enthalten, die restlichen Kacheln werden für die Analyse der Trace-Daten verworfen. Die Menge der übriggebliebenen Kacheln sei T. Um diese Kacheln zu ermitteln, verfügt das MOSM-Datenbank-Backend über eine PostGIS-Datenbank, die einen aktuellen Dump der OSM-Datenbank enthält. Über diese wird das Vorhandensein von POIs im Gebiet einer Kachel abgefragt. Das zurückgelieferte Ergebnis wird für zukünftige Abfragen in ”poi_tags“ eingefügt, da das Abfragen sämtlicher POIs eines Gebietes die Laufzeit der Abfrage wesentlich erhöht. Für die Mengen der in den Gebieten zweier aufeinanderfolgend besuchter Kacheln vorkommenden POI-Typen (diese seien U_1 und U_2) werden nun diejenigen Kanten in ”poi_graph“ inkrementiert, die in $U_1 \times U_2$ vorkommen.

5.1.2 Simulator

Der in C implementierte Simulator stellt folgende Funktionalität bereit:

Simulation von sich bewegenden Clients anhand von OSM-Traces:
Die auf ”http://www.openstreetmap.org/traces“ verfügbaren Traces spiegeln durch eine dichte Aneinanderreihung von Wegpunkten und dazugehörigen Zeitstempeln tatsächlich zurückgelegte Wege von echten Personen wider. Dadurch entfällt die Notwendigkeit eines Weggenerators oder ähnlicher Hilfsmittel zur Simulation von Personenbewegungen. Gleichzeitig sind so menschliche Bewegungsmuster gegeben, die nicht ohne Weiteres generierbar wären. Die o.g. Trace-Files liegen im GPS Exchange Format (GPX) vor, einem Format, das auf dem Extensible Markup Language (XML)-Datenformat basiert. Dieses Format dient in erster Linie dem Austausch von GPS-basierten Informationen zwischen Anwendungen und Webservices im Internet (siehe [top]).

Gezielte Auswahl von Trace-Files, die simuliert werden sollen: Anhand von Kriterien wie Start- und Zielpunkt, maximalem und minimalem erreichten Längen- und Breitengrad, Gesamtlänge, Durchschnittsgeschwindigkeit und Dauer des Traces kann spezifiziert werden, welche Traces als Grundlage für eine Simulation benutzt werden sollen.

Konvertierung einer Menge von Trace-Files in ein binäres Format: Zur Beschleunigung der Durchführung mehrerer Simulationen, die auf den gleichen Simulationsgrunddaten beruhen, können Trace-Dateien zusammengefasst und in einem binären Datenform gespeichert werden. Das erneute Lesen von Trace-Informationen wird so erheblich beschleunigt, da kein aufwändiges Parsen von XML-Dateien mehr notwendig ist.

Emulation von Hotspots in der Umgebung der Clients: Der Simulator verfügt über eine lokale PostgreSQL-Datenbank, in der Positionen generierter Hotspots abgelegt sind (siehe

6.1). Mit ihrer Hilfe werden virtuelle Clients in die Lage versetzt, abhängig von der Entfernung zum nächsten Hotspot, den virtuellen Internetkonnektivitätsstatus zu ändern.

Stellen von Map-Requests an den MOSM-Server: Falls sich ein virtueller Client in Reichweite eines virtuellen Hotspots befindet, wird ein Map-Request mit der aktuellen Position und dem Geschwindigkeitsvektor gestellt. Abhängig vom gerade getesteten Precachingverfahren kann auch der bisherige Trace des Clients übermittelt werden (nicht-geometrische Verfahren).

Logging von Cache Hits und Cache Misses: Über die virtuell zurückgelegte Wegstrecke des Clients werden für jede Zoomstufe Cache Hits und Cache Misses aufgezeichnet und gespeichert. Die Auswertung der Ergebnisdaten erfolgt in festen Längenabständen entlang des zurückgelegten Weges des Clients.

Permutieren von Trace-Files innerhalb eines Satzes von Simulationsgrunddaten: Nichtgeometrische Precachingverfahren verhalten sich beim Durchlaufen von Simulationen, basierend auf einer Menge von Trace-Files, in Abhängigkeit von der Reihenfolge, in der diese simuliert werden. Daher ist der Simulator in der Lage, verschiedene Sequenzen für die Abfolge von einzelnen Simulationen zu generieren und auszuführen. Diese Sequenzen sind wiederholt generierbar, sodass Messergebnisse jederzeit reproduzierbar sind.

5.2 Implementierung einer beispielhaften Nutzeranwendung mit Anbindung an die Openstreetmap-Datenbank

Unabhängig vom implementierten MOSM-Server und dem Simulator wurde eine Browser-Applikation unter Verwendung von PHP, Javascript, Gears und OpenLayers implementiert. Dank der in der OSM-API v0.6 neu hinzugekommenen Möglichkeit, vertrauenswürdigen Diensten über den OAuth-Mechanismus (siehe 4.3.5) teilweisen oder kompletten Zugriff auf geschützte API-Funktionalität zu geben, ist die Implementierung eines rudimentären mobilen OSM-Dienstes recht einfach möglich. Gears ermöglicht über seine LocalServer-API und Database-API die Offline-Funktionalität des Dienstes. Die Javascript-Bibliothek OpenLayers stellt lokal vorliegende Daten auf dem Client dar. Serverseitig sind über PHP die oben beschriebenen Precaching-Verfahren in den Dienst eingebunden.

Beispielhaft wurde folgende Funktionalität für diese Offline-Webapplikation implementiert:

- Anmeldung via OAuth und Erhalten eines OAuth Access Token
- Abrufen von Nutzerdaten
- Stellen eines Map-Requests an des MOSM-Dienst und senden der Kartenkacheln entsprechend des serverseitigen Precachingverfahrens
- lokale Speicherung der Kartenkacheln und Darstellen via Openlayers: Zur Zeit funktioniert das Speichern und Darstellen von Kartenkacheln nur mit dem Internetexplorer, da dieser ActiveX-Filesystemobjekte zum Lesen und Schreiben von Dateien auf dem lokalen Filesystem unterstützt. Da dies ein Sicherheitsrisiko im alltäglichen Umgang mit dem Browser darstellt, müssen für das Benutzen des MOSM-Dienstes die Sicherheitseinstellungen des Browsers heruntergesetzt werden, da sonst ein Teil der Funktionalität

der Webanwendung blockiert wird. Die Speicherung von Kartenkacheln ist prinzipiell auch in der durch Gears bereitgestellten lokalen Datenbank möglich. OpenLayers bietet allerdings bisher keine Möglichkeit, dies als Quelle für Bilddateien von Kartenkacheln zu wählen.

- Abrufen von POI-Informationen über die OSMXAPI
- Erstellen, Editieren, Löschen von POIs im Offline-Modus und Vormerken der Änderungen
- bei Bedarf Erstellen eines OSM-Changesets
- Übernahme der Änderungen in die OSM-Datenbank mit über OAuth autorisierten OSM-API-Zugriffen

Diese Implementierung ist als Ausblick und Anregung für eventuell weiterführende Projekte gedacht und an der in Kapitel 6 ausgeführten Evaluation nicht beteiligt. Der Anhang (Kapitel 7) enthält einige kommentierte Screenshots zu dieser Beispiel-Implementierung.

6 Evaluation

In diesem Kapitel sollen die vorgestellten Precachingverfahren für geografische Daten auf mobilen Clients evaluiert werden. Die Qualität der Verfahren ist durch die resultierende Cache Hit Rate beim Client

$$\frac{\#\text{Cache Hits}}{\#\text{Cache Hits} + \#\text{Cache Misses}}$$

und ihre Standardabweichung bestimmt. Je höher die Cache Hit Rate ist und je geringer die Standardabweichung, um so besser ist die Qualität des Precachingverfahrens einzustufen.

6.1 Simulationsaufbau

In diesem Abschnitt werden die durchgeführten Simulationen beschrieben sowie ihre Ergebnisse genannt und ausgewertet.

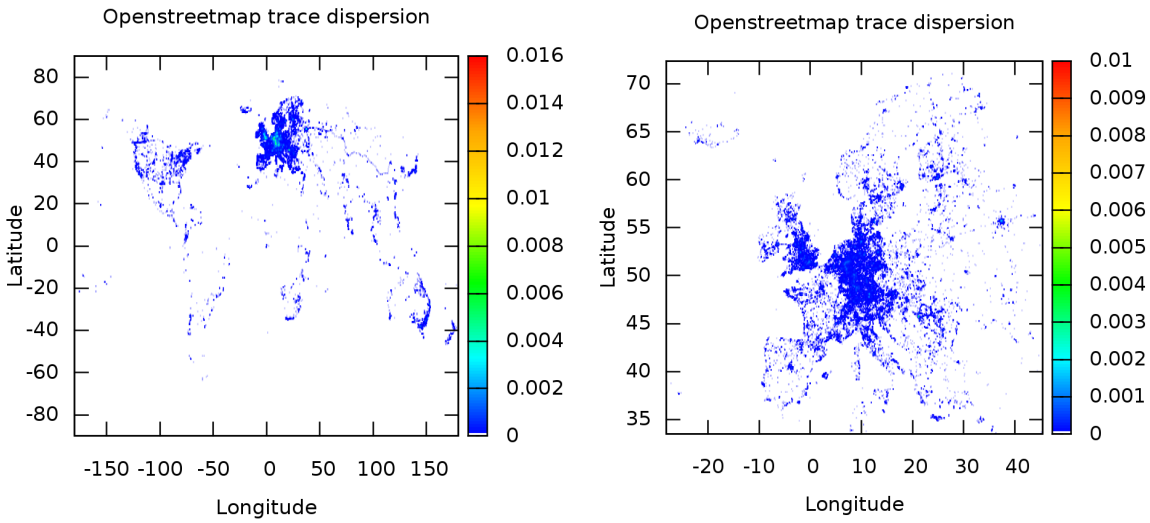
Die Simulationen sollen Aufschluss über die Qualität verschiedener serverseitig ausgeführter Precachingverfahren geben. Zu diesem Zweck wurde ein experimenteller Server (siehe Abschnitt 5.1.1) und ein Simulator (siehe Abschnitt 5.1.2) entwickelt und verschiedene Prechachingverfahren implementiert und getestet. Während der Simulator sowohl das Verhalten von Nutzern anhand von OSM-Traces als auch das Auftreten von Hotspots entlang eines Weges simuliert, berechnet der Server bei einer Anfrage eines virtuellen Clients unter Benutzung verschiedener Precachingverfahren (siehe 3.4.3) die Menge an zurückzuliefernden Kartenkacheln. Diese werden im virtuellen, lokalen und begrenzten Cache eines simulierten Clients gespeichert und, falls notwendig, nach der LRU-Verdrängungsstrategie (außerdem absteigend geordnet nach Zoomstufe und Entfernung zum aktuellen Standort) zugunsten neuer Kartenkacheln wieder aus dem Cache verdrängt. Es wird das Verhalten verschiedener Clients anhand von OSM-Traces, ausgewählt nach Gebiet bzw. anderen Eigenschaften, simuliert.

Das verwendete Simulationsgebiet S_1 ist begrenzt durch folgende Längen- und Breitengrade:

$$5,866 \leq lon \leq 15,042$$
$$47,270 \leq lat \leq 55,057$$

Diese bilden ein minimales Rechteck, das das Polygon “Deutschland” komplett enthält.

Dass diese Einschränkung sinnvoll ist, geht aus Abbildung 6.1 hervor. Sie veranschaulicht die Menge von OSM-Traces, die pro Flächeneinheit in einem Gebiet existieren, wobei sich über den Startpunkt eines Traces die Zugehörigkeit zu einem Gebiet ergibt. Um für ein Gebiet eine Aussage über die Qualität eines Precachingverfahrens treffen zu können, ist es wichtig, eine statistisch repräsentative Menge von Traces in diesem Gebiet nutzen zu können. Damit kommen nur Gebiete mit einer möglichst hohen Dichteverteilung von Traces als Simulationsgebiet in Frage.



(a) Weltweite Dichteverteilung

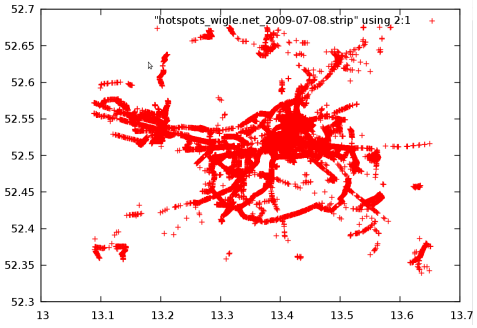
(b) Dichteverteilung für Europa

Abbildung 6.1: Dichteverteilung der OSM-Traces

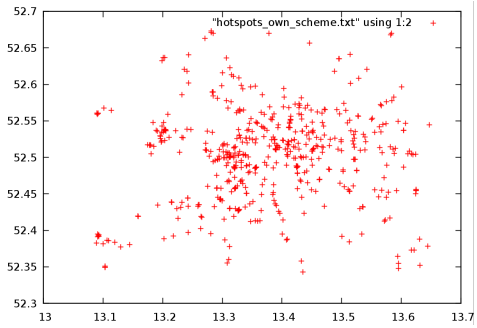
Verteilung der Hotspots

Für die durchgeführten Simulationen wurden Hotspot-Positionen für das oben definierte Simulationsgebiet generiert. Dies geschah nicht zufällig, sondern orientierte sich am Vorhandensein bestimmter POI-Typen (Tankstellen, Banken und Rathäuser) und Ihrer Position im Simulationsgebiet. Es ist plausibel anzunehmen, dass die Dichte derartiger Gebäude etwa proportional zu der von Hotspots in einem Gebiet ist. Auf diese Weise werden im gesamten Simulationsgebiet 23.157 Hotspots simuliert.

Eine Alternative zur beschriebenen Herangehensweise wäre die Benutzung der Hotspot-Datenbank Wigle (http://wigle.net). Hierbei handelt es sich um ein Wiki-Projekt, das die weltweite Katalogisierung von Hotspots zum Ziel hat. Die Datenbank enthält u.a. die Position (lon,lat), Service Set Identifier (SSID) und Basic Service Set Identifier (BSSID) einzelner Hotspots sowie den Zeitstempel der letzten Aktualisierung des jeweiligen Eintrags in der Wigle-Datenbank. Eine Visualisierung der in dieser Datenbank eingetragenen Hotspots von Berlin (Abbildung 6.2) verdeutlicht jedoch durch diverse unerschlossene “weiße Flächen” die Unvollständigkeit der Datenerhebungen. Somit wurde hier zugunsten der Nachvollziehbarkeit und der Plausibilität die oben beschriebene generische Lösung angewendet und fiktive Hotspots generiert.



(a) Hotspotverteilung der Datenbank "Wigle" für Berlin



(b) Generierte Verteilung von virtuellen Hotspots für Berlin

Abbildung 6.2: Hotspotverteilung für Berlin

Reichweite der einzelnen simulierten Hotspots Die hier durchgeführte Simulation legt eine WLan-Abdeckung jedes einzelnen Hotspots in einem Radius von 300 Metern zugrunde. Diese Annahme ist von der Realität natürlich weit entfernt. Unter günstigen Bedingungen können hier ca. 50 Meter angenommen werden. Um jedoch die Anzahl der zustandekommenden Internetverbindungen eines virtuellen Clients beim Abschreiten seines simulierten Weges zu erhöhen, wird in dieser Simulation für jeden einzelnen Hotspot von einer größeren Reichweite ausgegangen als sie in der Realität anzunehmen ist.

6.2 Bestimmung der Qualität des serverseitigen Precachings

Um die Qualität und somit die Anwendbarkeit der in den vorigen Kapiteln eingeführten Precachingverfahren zu ermitteln, werden verschiedene Simulationen mit verschiedenen Mengen von OSM-Traces durchgeführt, deren jeweilige Elemente entweder in demselben Simulationsgebiet liegen oder eine andere Eigenschaft, wie z.B. ihre Länge, gemeinsam haben. Als Vergleichsreferenz dient bei jeder Simulation das kreisförmige Precaching, welches das einfachste Verfahren darstellt. Das Planquadrat, das die Stadt Berlin umschließt, stellt das Teilsimulationsgebiet S_2 dar. Hierfür gilt:

$$13,088 \leq lon \leq 13,658$$
$$52,338 \leq lat \leq 52,675$$

Innerhalb dieses Bereichs wurden Simulationen für alle geometrischen und nicht-geometrischen Precachingverfahren durchgeführt. Elliptische Verfahren wurden mit $f = \frac{A}{B} \in \{2, 3, 4\}$, kreissektorförmige mit $\varphi \in \{90°, 135°, 180°\}$ durchgeführt. Abbildung 6.3 zeigt die Cache Hit Rate, gemittelt über alle 623 in Berlin simulierten Traces.

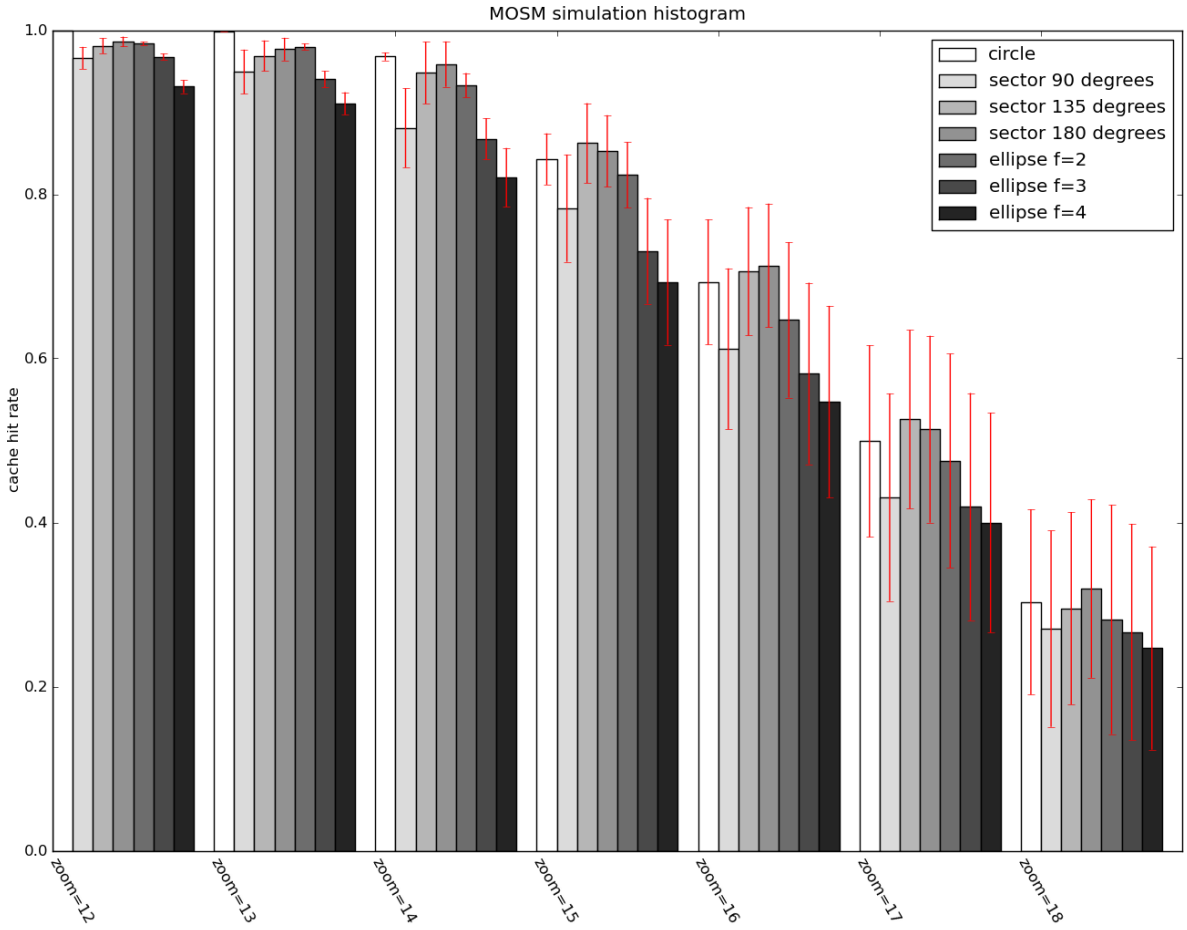


Abbildung 6.3: Cache Hit Rate für geometrisches Caching für den Bereich Berlin, sortiert nach Zoomstufen (Standardabweichung: rote Fehlerbalken)

Die Abbildung 6.3 zeigt die gemessene mittlere Cache Hit Rate sowie ihre Standardabweichung und beschränkt sich auf die Darstellung der Zoomstufen 12 bis 18, da die Werte der Cache Hit Rate für darunter liegende Zoomstufen nahe oder genau bei 100% liegen. Die großen Gebiete, die von Kartenkacheln dieser niedrigen Zoomstufen abgebildet werden, werden von Clients kaum verlassen, wodurch die Cache Hit Rate immer konstant hoch oder maximal ist. Deshalb enthalten die Messungen für die unteren Zoomstufen keine interessanten Informationen. Auffallend an dieser Grafik ist, dass das kreisförmige Caching ab der Zoomstufe 15 kaum schlechter ist als das hier beste Verfahren, das kreissektorförmige Caching mit einem Mittelpunktswinkel von 180°, und sich auch die Standardabweichungen kaum unterscheiden. Für die Zoomstufen kleiner oder gleich 14 ist das kreisförmige Precaching sogar optimal. Die Cache Hit Rate des elliptischen Precachings ist insgesamt durchgängig niedriger als die des kreissektorförmigen Precachings. Dies ist im Nachhinein plausibel, da die Ellipse nach "vorn" immer schmaler wird, der Kreissektor im Vergleich jedoch breiter entsprechend der mit zunehmender Entferung zum aktuellen Standort immer unsicherer werdenden Prognose des Precachingalgorithmus. Das hier ermittelte Ergebnis deckt sich qualitativ vollständig und quantitativ mit geringen Abweichungen mit der Erkenntnis aus Simulationen für Traces der Länge 30 km bis

40 km im gesamten oben spezifizierten Simulationsgebiet S_1. Diese Simulation umfasst 3150 Traces.

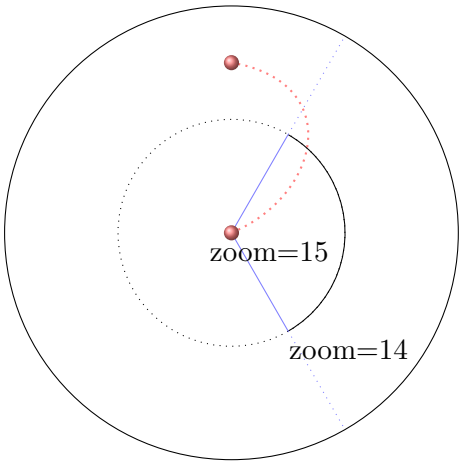


Abbildung 6.4: Verhalten des kreissektorförmigen Precaching bei niedriger werdenden Zoomstufen

Die gute Qualität des kreisförmigen Precachings im Vergleich zu den restlichen geometrischen Verfahren für Zoomstufen ≤ 14 ist bedingt durch die Ungenauigkeit der Vorhersage der Bewegung eines Clients für die "ferne Zukunft". Abbildung 6.4 veranschaulicht dies. Während in diesem Beispiel das kreissektorförmige Precaching für die Zoomstufe 15 noch ein qualitativ gutes Ergebnis liefert, verläuft die Trajektorie des Clients für die nächst niedrigere Zoomstufe 14 aus dem durch das Precaching gegebenen Kreissektor heraus, womit an dieser Stelle ein kreissektorförmiges Precaching mit größerem Mittelpunktswinkel bzw. das kreisförmige Precaching bessere Ergebnisse liefern.

Abbildung 6.5 zeigt die Cache Hit Rate für das Precaching, basierend auf Potentialen, sowie die Cache Hit Rate, basierend auf der Ähnlichkeit von Umgebungen. Wie schon in Abschnitt 3.4.3 beschrieben, wird beim nicht-geometrischen Precaching der durch den Mittelpunktswinkel definierte Kreissektor (hier: 90°, 135°, 180°) unter Benutzung der Kachelgewichte effektiv verformt. Die in der Abbildung gezeigten Messresultate basieren auf je zwei Simulationsdurchläufen pro Precachingverfahren und Mittelpunktswinkel. Der erste (mit *build phase* in der Abbildung 6.5 bezeichnet) sorgt für das Aufbauen einer Potentialverteilung bzw. eines POI-Graphen. Der zweite Durchlauf (mit *use phase* in der Abbildung 6.5 bezeichnet) benutzt die Potentialverteilung bzw. den POI-Graphen, um den letztendlichen Effekt des jeweiligen Verfahrens zu ermittlen. In allen Fällen liegt die resultierende Cache Hit Rate unter der des rein geometrischen Precachingverfahrens. Beim Precaching, basierend auf Potentialen, ist ein leichter Anstieg der Cache Hit Rate in der *use phase* im Vergleich zur *build phase* zu erkennen. Die verbesserte Cache Hit Rate liegt jedoch immer noch unter der des kreisförmigen Cachings. Dieser leichte Effekt dürfte bei der Simulation jedes einzelnen Traces lediglich auf die in der *use phase* selbsterzeugten Kachelgewichte zurückzuführen sein. Von anderen Traces gesetzte Gewichte scheinen bei der Anwendung des Precachingverfahrens eher störend zu wirken.

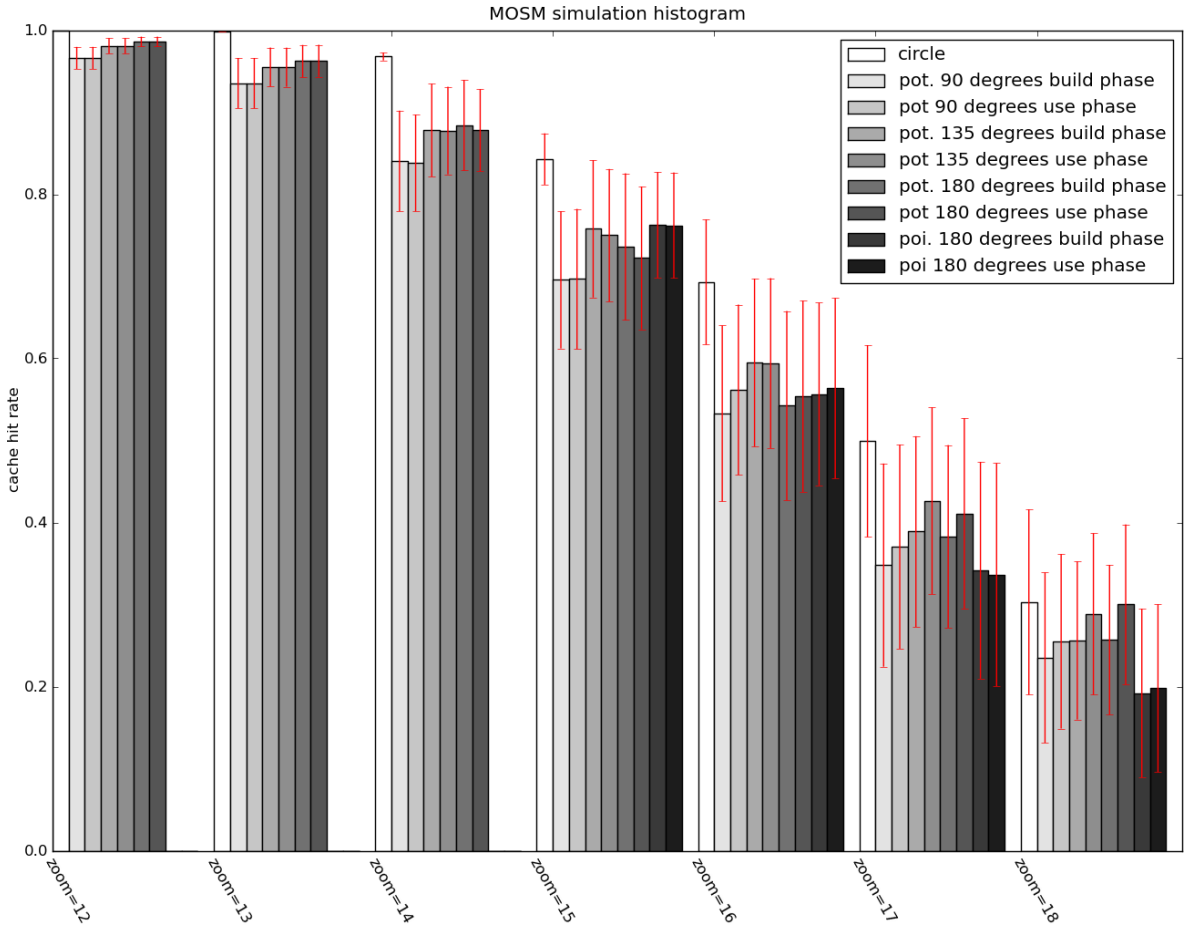


Abbildung 6.5: Cache Hit Rate für nicht-geometrisches Caching für den Bereich Berlin, sortiert nach Zoomstufen (Standardabweichung: rote Fehlerbalken)

Die möglichen Gründe für das unerwartete und unerwünschte Verhalten beider nicht-geometrischer Verfahren sind vielfältig:

- Das durch die OSM-Traces widergespiegelte Nutzerverhalten entspricht nicht dem tatsächlichen durchschnittlichen Bewegungsprofil von Menschen.
- Die hier verfügbare Anzahl von Traces pro Fläche ist unzureichend, um komplexere Bewegungsmuster von Menschen zu erfassen, d.h., um das Erstellen einer brauchbaren Verteilung von Gewichten über Kartenkacheln bzw. das Erstellen eines brauchbaren POI-Graphen zu ermöglichen.
- **spezifisch für das Precaching, basierend auf der Ähnlichkeit von Umgebungen:** Die Entscheidung, ob ein Client tatsächlich einen POI besucht hat, lässt sich nach verschiedenen Kriterien treffen. Das treffsicherste Kriterium wäre eine Aussage des Nutzers selbst, welche hier allerdings nicht verfügbar ist. Weitere Kriterien sind die Entfernung zum POI, die Geschwindigkeit beim Passieren des POI oder die Dauer des Aufenthalts am POI. So kann ein Objekt wie das Brandenburger Tor nur durch Vorbeilaufen als "besucht" gelten, für ein Restaurant würde das Analoge sicher nicht gelten. Je

nach Art des POI lassen diese Kriterien also verschiedene Schlußfolgerungen über einen tatsächlich stattgefundenen "Besuch" eines POI zu, was in jedem Fall jedoch eine Abschätzung mit Wahrscheinlichkeitscharakter bleibt. Aus Gründen der für diese Arbeit beschränkt verfügbaren Rechenkapazität[1] wurde als Kriterium die Nähe eines Clients zu dem jeweiligen POI verwendet. "Nähe" bedeutet in der Simulation, dass Client und POI sich auf derselben Kachel befinden. Trotz dieser sehr approximativen Herangehensweise ist die tatsächliche Simulation sehr langwierig, da sie durch SELECT-Anweisungen auf der ca. 11 GB großen PostGIS-Datenbank ausgebremst wird. Es ist zu vermuten, dass die Anwendung weiterer Kriterien die Cache Hit Rate verbessert, jedoch die Laufzeit einer Anfrage weiter erhöht.

Um die Qualität des kreissektorförmigen Precachings differenzierter zu untersuchen, werden im Weiteren Simulationsmessergebnisse, die mit nach "Geradheit" sortierten Traces durchgeführt wurden, ausgewertet.

Definition "Geradheit": Die "Geradheit" eines Traces bzw. des von ihm beschriebenen Weges wird hier als Verhältnis zwischen Abstand von Start- und Zielpunkt und der Länge des Weges entlang seiner Trajektorie definiert:

$$d = \frac{|\vec{P}_{Start} - \vec{P}_{End}|}{\int_{\vec{P}_{Start}}^{\vec{P}_{End}} ds}; \qquad ds = \sqrt{dx^2 + dy^2} = dt\sqrt{\left(\frac{dx}{dt}\right)^2 + \left(\frac{dy}{dt}\right)^2}$$

Der Weg von $\vec{P}_{Start}$ bis $\vec{P}_{End}$ wird durch die kartesischen Koordinaten $x(t), y(t)$ parametrisiert (t ist der Wegparameter, $t_{Start} \leq t \leq t_{End}$).
Mit zunehmender Geradheit sollte auch die Qualität des kreissektorförmigen Precachings steigen: Bei einer Geradheit von $d = 1$ bildet der Trace eine Gerade, womit jede Art von geometrischem Precaching in Bewegungsrichtung voraus ihre optimale Qualität liefert. Je mehr sich die Geradheit an den Wert 0 annähert, desto geringer sollte statistisch auch die Qualität des Precachings sein. Dies wurde für Traces der Geradheit $d = 0.1 - 1.0$ in Intervallen von 0.1 und einer Länge von 10 km bis 20 km getestet. Für jedes Geradheitsinterval wurde die Simulation mit jeweils 50 zufällig ausgewählten Traces der entsprechenden Geradheit aus dem Simulationsgebiet S_1 benutzt. Jede Simulation wurde für kreissektorförmiges Precaching mit Mittelpunktswinkeln von 40°-360° (360°=kreisförmig) in 20°-Abständen durchgeführt. Eine Zeile der Legende der Abbildungen 6.6 und 6.7 ist als Tupel (Mittelpunkswinkel, Geradheit der simulierten Traces) zu lesen. Aus dem Ergebnis der Simulationen geht, den Erwartungen entsprechend, hervor, dass sich kreisförmiges Caching zum zunehmender Geradheit von Bewegungen mehr und mehr lohnt. Dies wird in den Abbildungen 6.6 und 6.7 für die Geradheiten d=0.6 und d=0.8 dargestellt.

[1] Intel(R) Core(TM)2 Quad CPU @ 2.40GHz, 8GB RAM, Ubuntu Linux 64 Bit 9.04

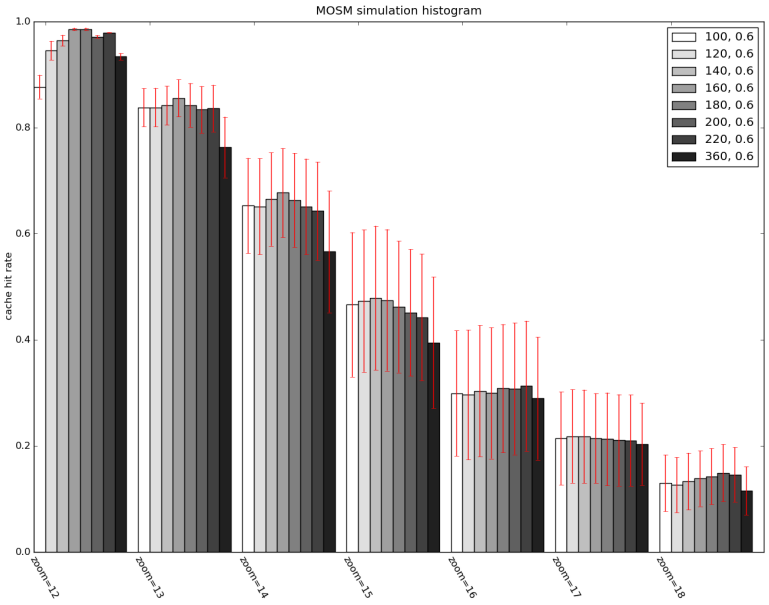


Abbildung 6.6: Cache Hit Rate für Geradheit $d = 0,6$ (Standardabweichung: rote Fehlerbalken)

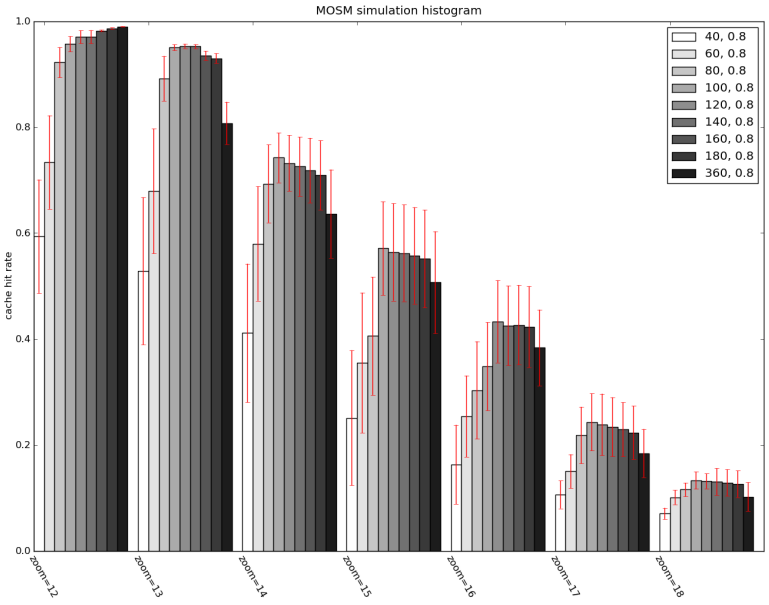


Abbildung 6.7: Cache Hit Rate für Cache Hit Rate für Geradheit $d = 0,8$ (Standardabweichung: rote Fehlerbalken)

Aus den Abbildungen 6.6 und 6.7 ist ein tendenzieller Anstieg des Verhältnisses der Cache Hit Rate zwischen kreissektorförmigem und kreisförmigem Precaching ablesbar. Demnach realisiert sich der Vorteil des kreissektorförmigen Precachings gegenüber dem kreisförmigen

Precachings mit steigender Geradheit der Traces. Dies wird zusätzlich durch Abbildung 6.8 für die Zoomstufe 15 illustriert. Oberhalb der Werte d=0.6 zeigt sich eine nicht zu vernachlässigende Verbesserung des kreissektorförmigen Precachings gegenüber dem kreisförmigen Fall.

Abbildung 6.8: Verhältnis der maximalen Cache Hit Rate (CHR) bei kreissektorförmigem Precaching zu der bei kreisförmigem Precaching als Funktion der Geradheit d von Traces

6.3 Messung zum Datenvolumen im Netzwerk

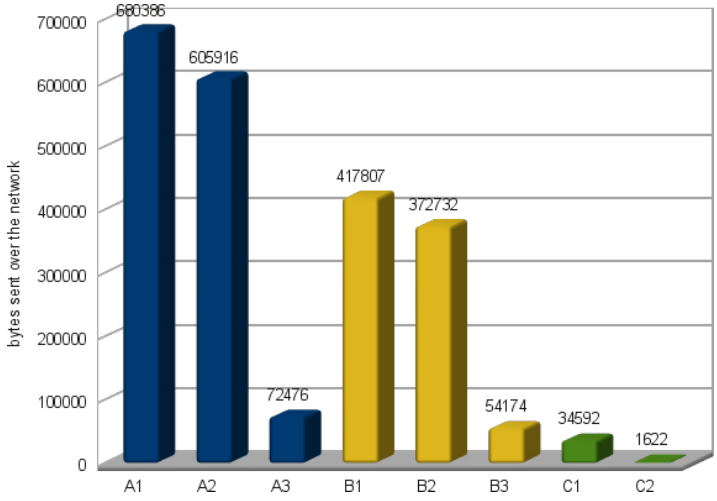


Abbildung 6.9: Netzwerk-Datenaufkommen beim Herunterladen von OSM-Daten

Abbildung 6.9 zeigt das benötigte Datenvolumen zur Übertragung von Kartenkacheln und der Übertragung der XML-Repräsentation der in den Kartenkacheln enthaltenen Informationen. Diese Informationen sind z.B. für das Suchen nach geografischen Objekten oder auch für das

Editieren derselben notwendig. Die durchgeführte Messung wurde für das Gebiet einer Kachel der Zoomstufe 15 (entspricht an dieser Stelle einer abgebildeten Fläche von ca. 750 m × 750 m) durchgeführt[2]. Die verschiedenen Balken im Diagramm der Abbildung 6.9 haben folgende Bedeutung:

- **A1:** Das Downstream-Datenvolumen (Payload + Protocol Overhead) beim Herunterladen der oben spezifizierten Kartenkachel der Zoomstufe 15 als Bilddatei und der Kartenkacheln höherer Zoomstufen bis Stufe 18, die das Gebiet dieser Kachel abdecken ($= \sum_{i=0}^{3} 2^{2 \cdot i} = 85$ Kartenkacheln)
- **A2:** = A1 ohne Protocol Overhead
- **A3:** Das Upstream-Datenvolumen beim Herunterladen der oben spezifizierten Kartenkachel der Zoomstufe 15 als Bilddatei und der Kartenkacheln höherer Zoomstufen bis Stufe 18, die das Gebiet dieser Kachel abdecken
- **B1:** Das Downstream-Datenvolumen (Payload + Protocol Overhead) beim Herunterladen aller Kartenkacheln der Zoomstufe 18, die zusammen das Gebiet der oben beschriebenen Kartenkachel der Zoomstufe 15 abdecken ($= 2^6 = 64$ Kartenkacheln)
- **B2:** = B1 ohne Protocol Overhead
- **B3:** Das Upstream-Datenvolumen beim Herunterladen aller Kartenkacheln der Zoomstufe 18, die zusammen das Gebiet der oben beschriebenen Kartenkachel der Zoomstufe 15 abdecken
- **C1:** Das Downstream-Datenvolumen beim Herunterladen der XML-Repräsentation der Informationen der in den bei A1 heruntergeladenen Kacheln enthaltenen Informationen über die OSM-API (siehe 4.3)
- **C2:** Das Upstream-Datenvolumen beim Herunterladen der XML-Repräsentation der Informationen der in den bei A1 heruntergeladenen Kacheln enthaltenen Informationen über die OSM-API

Dies verdeutlicht anschaulich die Größenordung der zu transportierenden Datenvolumina, die notwendig sind, um OSM als mobilen Kartendienst zu nutzen, wobei sämtliche Daten bei Bedarf bzw. im Voraus heruntergeladen werden. Um Kartenkacheln eines Gebiets von 7,5 km × 7,5 km in den Zoomstufen 15 bis 18 herunterzuladen (dies ist für den praktischen Anwendungsfall immer noch ein sehr eingeschränktes Gebiet), ist überschlagsmäßig auf der Basis der in Abbildung 6.9 dargestellten Messung mit einem Gesamtdatenvolumen von ca. 70 Megabytes zu rechnen. Das Datenvolumen beim Herunterladen der XML-Repräsentation der Karteninformationen beschränkt sich hierbei auf ca. 3,6 MB. Bei einer optimistischen Abschätzung der verfügbaren Datenrate von Hotspots von 20Mbit/s wird eine minimale Verbindungszeit von ca. 28 Sekunden benötigt, um 70 Megabytes an Daten herunterzuladen. Da diese Datenrate im Praxisfall nur in Einzelfällen verfügbar ist und ebenso eine Verbindungsdauer von 28 Sekunden an einem Hotspot aufgrund der Natur des Anwendungsfalls nicht ohne Weiteres anzunehmen ist, ist es für den Betrieb des Dienstes sinnvoll, im Voraus

[2]Es wurde diejenige Kachel der Zoomstufe 15 gewählt, die den Campus der Humboldt-Universität zu Berlin in Berlin-Adlershof abbildet.

einen gewissen Grunddatenstamm an Kartenmaterial lokal zu bevorraten. Somit müssten im laufenden Betrieb nur Kartenkacheln, die aktuelle Änderungen enthalten, zum Client übertragen werden. Das gilt auch für die XML-Repräsentation der in den Kartenkacheln enthaltenen Informationen. Zur Erstellung des Grunddatenstammes ist allerdings ein Eingreifen des Nutzers notwendig, um die Auswahl des demnächst besuchten Gebietes zu treffen. Dies erfordert jedoch einen Kompromiss zwischen der Minimierung der Nutzerinteraktion mit dem Dienst, um dessen volle Funktionalität zu gewährleisten und der Einschränkung des Datenverkehrs im mobilen Einsatz durch eine Vorselektierung von Kartendaten durch den Nutzer. Eine weitere Möglichkeit wäre das eigenständige Rendern von Karten durch den Client, sodass der Download von Bilddateien entfällt. Dies jedoch würde bei mobilen Geräten die Batterielaufzeit wesentlich verkürzen. Im Zuge der sich entwickelnden Energiespeichertechnologien und der immer kleiner werdenden Baugrößen von Prozessoren mit immer geringerem Stromverbrauch könnte dies jedoch zunehmend eine Option werden.

6.4 Fazit und Ausblick

Das Ziel des in dieser Arbeit vorgestellten geografischen Kartendienstes ist das Versorgen eines in unvorhersehbaren Intervallen zum Server verbundenen mobilen Clients mit Kartenmaterial. Diese Anforderungen führen zu einem Design des Dienstes als LBS und Offline-Webanwendung. Über die Nutzung von in der Vergangenheit durchgeführten Beobachtungen der Bewegungen des Anfrage stellenden Clients oder (abhängig vom Precachingverfahren) anderer Clients wird versucht, die Auswahl des geografischen Gebietes, für das Kartenmaterial zum Client gesendet wird, zu optimieren. Verschiedene plausible Ansätze wurden dabei im Verlauf dieser Arbeit beschrieben und evaluiert. Von allen in Betracht gezogenen Precachingverfahren konnte das kreissektorförmige Precaching in einigen Fällen eine bessere Ergebnisqualität liefern als das kreisförmige Verfahren. Im Gegensatz dazu konnte das elliptische Precaching keine Verbesserung erzielen. Auch die Anwendung der nicht-geometrischen Precachingverfahren führte zu keiner Verbesserung im Vergleich zu kreisförmigem Precaching. Inwieweit dies bedingt ist durch die Spezifik der benutzten, von OSM zur Verfügung gestellten Traces als Simulationsgrunddaten ist unklar und bedarf weiterer Untersuchungen mit Trace-Datenmaterial mit nachweisbarer statistischer Relevanz. Die weiterhin vorgenommene Messung und Hochrechnung des benötigten Netzwerk-Datenaufwands für den Betrieb eines auf Openstreetmap basierenden LBS verdeutlicht die Probleme, die unabhängig vom Precaching des Dienstes existieren. Bei der Versorgung eines Clients mit Kartenmaterial über sporadisch auftretende Internetverbindungen an z.B. öffentlichen Hotspots kann bei zu seltenen und zeitlich zu kurz andauernden Verbindungen zum Internet die vollständige Funktionalität des Dienstes nicht mehr gewährleistet werden.

Eine logische Weiterführung der Arbeit wäre der Ausbau der bestehenden Implementierung hin zu einer Offline-Webanwendung, die einen Großteil der gängigen Funktionalität von Openstreetmap beinhaltet. Hierbei sollte überprüft werden, ob eine dynamische Partitionierung der Funktionalität des Dienstes zwischen Server und Client die Nutzbarkeit des Dienstes verbessern kann. Vorstellbar wäre, wie schon in Abschnitt 3.2 erwähnt, das Rendern einer Teilmenge von Kacheln auf dem Client. Dies könnte z.B. bei einer Verbindung zum Server mit einer schlechten Datenrate das herunterzuladene Datenvolumen senken und so die Funktionalität des Dienstes im Offlinemodus auch bei einer geringeren Verbindungsdauer zum Server gewährleisten.

Eine weitere Herausforderung besteht u.a. in dem Design einer sinnvollen Oberfläche für mobile Geräte, die z.B. komplexere Editiervorgänge von Kartenmaterial erlaubt (zur Veranschaulichung, siehe Screenshot der entsprechenden Desktop-Anwendung *JOSM* im Anhang, Abbildung 7.6). Die sehr beschränkte Displaygröße von z.B. Smartphones ist hierfür ein besonders limitierender Faktor.

7 Anhang

Abbildung 7.1: kreissektorförmiges Precaching im Anwendungsfall

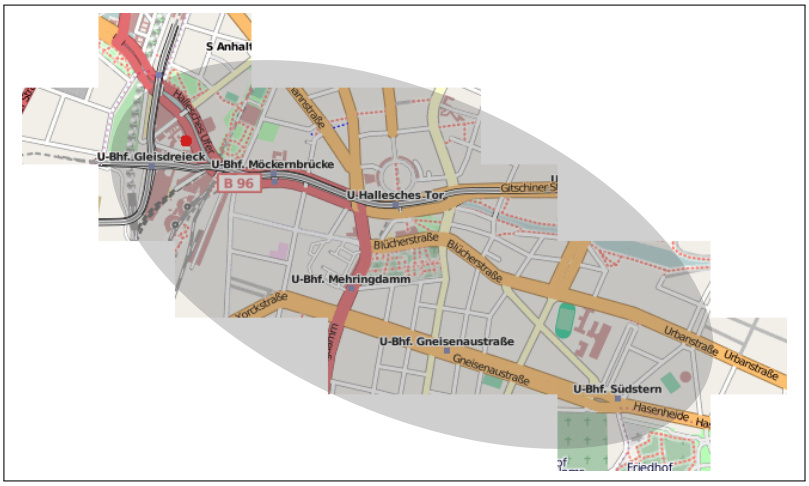


Abbildung 7.2: elliptisches Precaching im Anwendungsfall

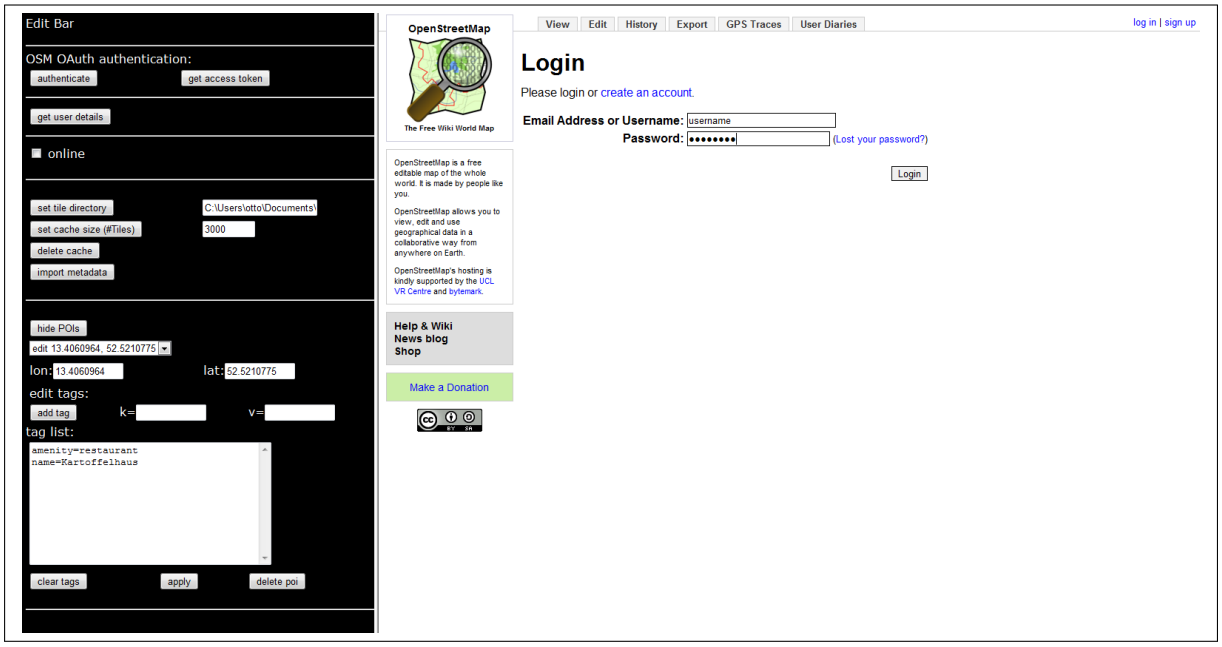


Abbildung 7.3: MOSM Browseranwendung: OAuth Authorisierung; Bevor schreibende Funktionen der Openstreetmap-API von einem Webservice (Consumer) benutzt werden können, muss dieser autorisiert werden. Nach der Authentifikation des Nutzers wird die URL des Frameinhalts zur "Callback-URL" gesendet, der URL der Consumer-Website. Diese fordert daraufhin ein Access-Token vom Service-Provider an.

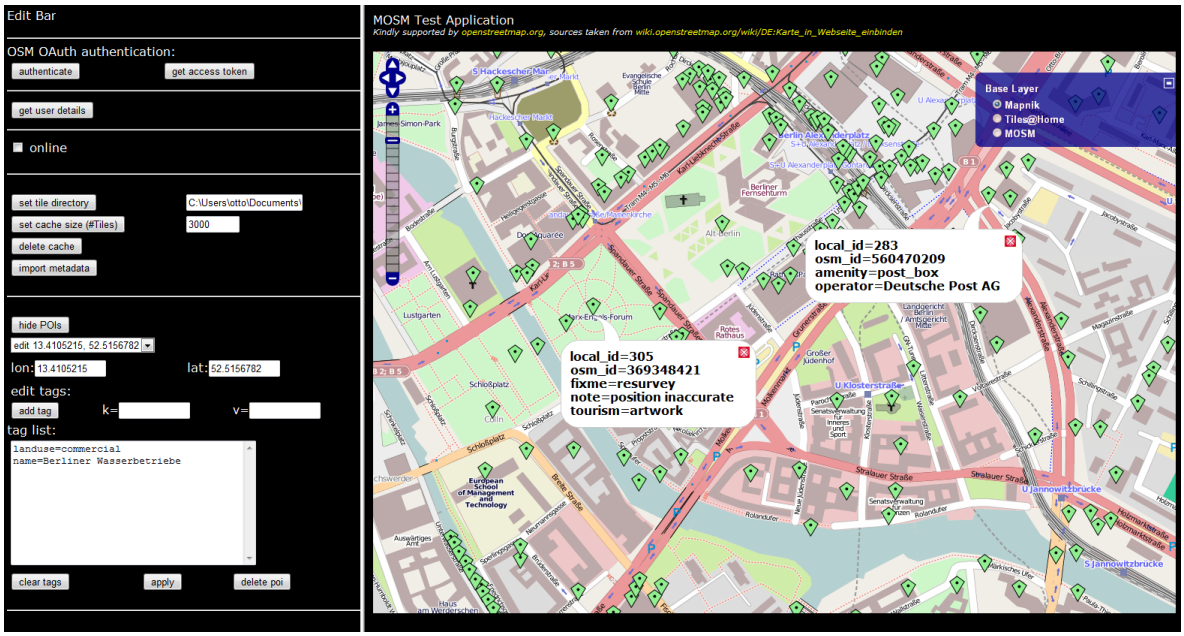


Abbildung 7.4: MOSM Browseranwendung: Ansicht und Bearbeiten von POIs; Die gezeigt Webanwendung beherrscht grundlegende Funktionalität wie das Setzen und Bearbeiten von Points of Interest. Diese werden als thematische Schicht über der Kartenansicht angezeigt. Änderungen werden übertragen, sobald die Anwendung in den Online-Status übergeht. Der Einfachheit halber wird eine Statusänderung der Internetverbindung hier über die Online-Checkbox durch den Nutzer kommuniziert.

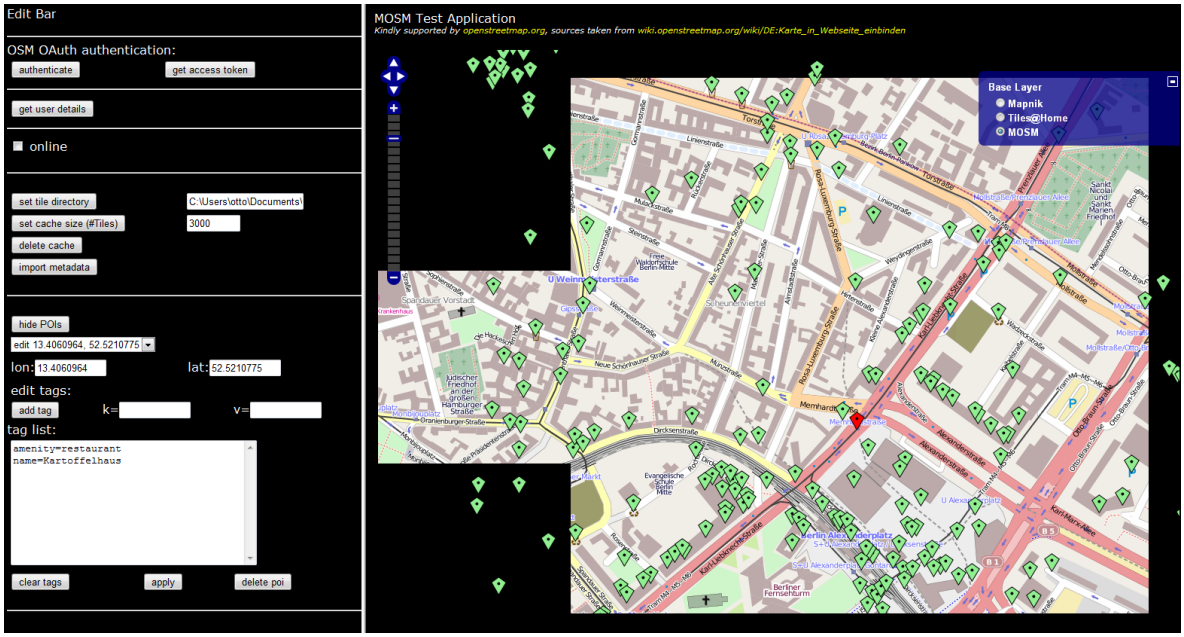


Abbildung 7.5: MOSM Browseranwendung: Ansicht von Kartenkacheln in lokalem Cache; Das in der Webanwendung verwendete Openlayers unterstützt die Darstellung von Kartenkacheln aus verschiedenen Quellen wie z.B. dem Openstreetmap-Tile-Server, Tiles@Home und auch dem lokalen Filesystem, auf dem der Client der Webanwendung Kacheln ablegt. Openlayers zeigt hier einen Teil jener Kartenkacheln an, die vom Precachingverfahren des MOSM-Service zurückgeliefert wurden.

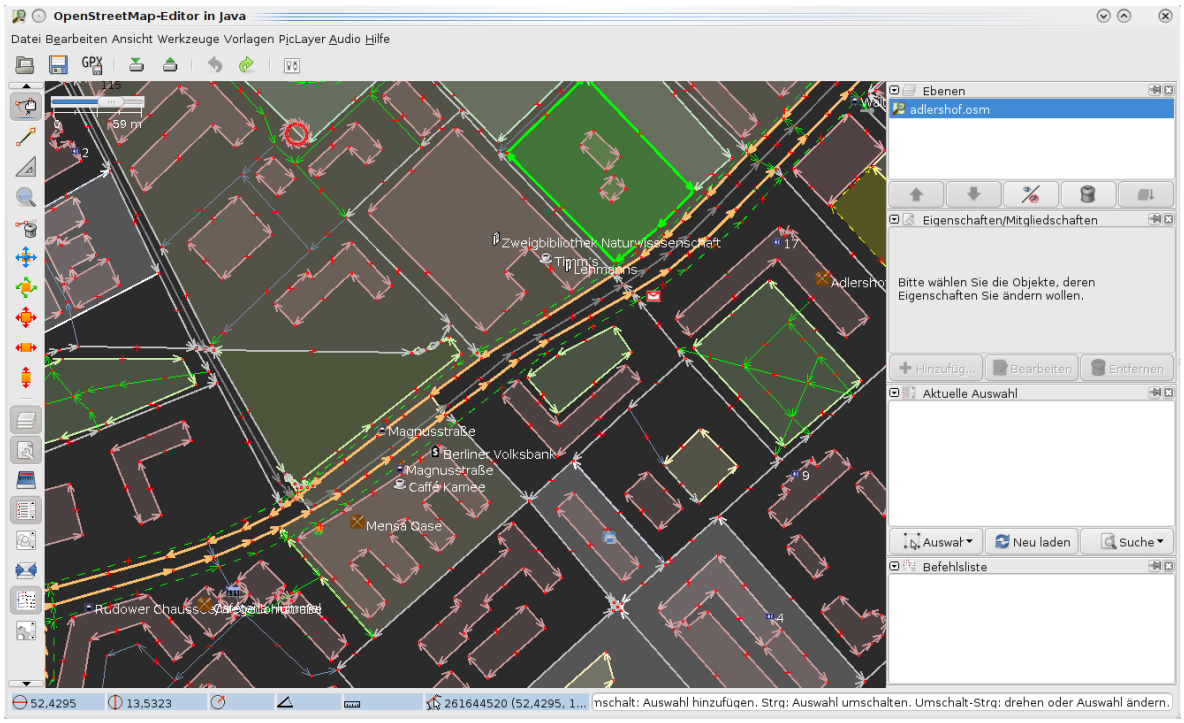


Abbildung 7.6: Screenshot des Karteneditors JOSM

Abkürzungsverzeichnis

AJAX *Asynchronous JavaScript and XML.*

AOA *Angle if Arrival.*

API *Application Programming Interface.*

ASP *Active Server Pages.*

BSSID *Basic Service Set Identifier.*

CGI *Common Gateway Interface.*

COA *Cell of Origin.*

DDL *Data Definition Language.*

DOM *Document Object Model.*

ECMA *European Computer Manufacturers Association.*

EWKB *Extended Well Known Binary.*

EWKT *Extended Well Known Text.*

FIFO *First In - First Out.*

GIS *Geographisches Informationssystem.*

GPS *Global Positioning System.*

GPX *GPS Exchange Format.*

GSM *Global System for Mobile Communications.*

HTML *Hypertext Markup Language.*

HTTP *Hypertext Transfer Protocol.*

JSON *JavaScript Object Notation.*

LAN *Local Area Network.*

LBS *Location Based Service.*

LRU *Least Recently Used.*

MOSM *Mobile Openstreetmap.*

OGC *Open Geospatial Consortium.*

OSM *Openstreetmap.*

OSMXAPI *OSM Extended API.*

PDA *Personal Digital Assistant.*

PKI *Public Key Infrastructure.*

POI *Point of Interest.*

SSI *Server Side Includes.*

SSID *Service Set Identifier.*

TCL *Tool Command Language.*

TDOA *Time Difference of Arrival.*

UMTS *Universal Mobile Telecommunications System.*

USB *Universal Serial Bus.*

WFS *Web Feature Service.*

WKB *Well Known Binary.*

WKT *Well Known Text.*

WMS *Web Map Service.*

XML *Extensible Markup Language.*

Abbildungsverzeichnis

Literaturverzeichnis

[Arn04] Arnaud Le Hors (IBM), Philippe Le Hégaret (W3C), Lauren Wood (SoftQuad, Inc.), Gavin Nicol (Inso EPS), Jonathan Robie (Texcel Research and Software AG), Mike Champion (Arbortext and Software AG), Steve Byrne (JavaSoft) . Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/, April 2004.

[car] Cartography for swiss higher education (cartouche) - module 'location based services'. http://www.e-cartouche.ch/overview.php.

[cre] creative commons. http://creativecommons.org/licenses/by-sa/2.0/.

[Dav99] Dave Raggett (W3C),Arnaud Le Hors (W3C), Ian Jacobs (W3C). HTML 4.01 Specification. http://www.w3.org/TR/html4, December 1999.

[Dem09] Michael N. Demers. *Fundamentals of Geographic Information Systems*. Wiley, 4th ed. edition, 2009.

[ea99] Jin Jing et al. Client-server computing in mobile environments. *ACM Computing Surveys*, 31(2), June 1999.

[geaa] Google gears api reference. http://code.google.com/intl/de-DE/apis/gears/api_summary.html.

[geab] Google gears faq. http://code.google.com/intl/de-DE/apis/gears/gears_faq.html#whatIsGears.

[GSM03] GSM Association. *Location Based Services, Version 3.1.0*, January 2003.

[Her06] John R. Herring. *OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, Version 1.2*. Open Geospatial Consortium Inc., October 2006.

[jse02] Information technology — ECMAScript language specification, Second Edition. http://standards.iso.org/ittf/PubliclyAvailableStandards/c033835_ISO_IEC_16262_2002(E).zip, June 2002.

[jsm] Core javascript 1.5 reference. https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference.

[Mar07] Mark Atwood et al. OAuth Core 1.0. http://oauth.net/core/1.0, December 2007.

[Mar09] Mark Atwood et al. OAuth Core 1.0 Revision A. http://oauth.net/core/1.0a, June 2009.

[MM08] ESRI Marwa Mabrouk. *OpenGIS Location Services (OpenLS): Core Services, Version 1.2*. Open Geospatial Consortium Inc., September 2008.

[off04] Smart client offline application block. http://msdn.microsoft.com/en-us/library/ms998460.aspx, 2004.

[osm] Openstreetmap api v0.6. http://wiki.openstreetmap.org/index.php/OSM_Protocol_Version_0.6.

[pos] Postgis 1.4.0 manual. http://postgis.refractions.net/download/postgis-1.4.0.pdf.

[R. 99] R. Fielding (UC Irvine), J. Gettys (Compaq/W3C), J. Mogul (Compaq), H. Frystyk (W3C/MIT), L. Masinter (Xerox), P. Leach (Microsoft), T. Berners-Lee (W3C/MIT). RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html, June 1999.

[top] Topografix: The gpx exchange format. http://http://www.topografix.com/gpx.asp.