Bachelorarbeit

Caching von HTTP-Range-Requests

von Simon Moser

5. Februar 2018

Albert-Ludwigs-Universität
Freiburg

Institut für Informatik
Lehrstuhl für Kommunikationssysteme

Erklärung

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

Ort, Datum	Unterschrift

Zusammenfassung

Die „Emulation as a Service“-Infrastruktur bietet die Möglichkeit, archivierte Software zu emulieren. Die benötigten Dateien (Images) werden dazu aus sogenannten Image Archives geladen. Bisher werden dabei mehrmals angefragte Imageblöcke auch wiederholt übertragen, was den Emulator verlangsamt. In dieser Arbeit geht es darum, das Laden aus entfernten Image Archives zu beschleunigen, indem ein Cache implementiert wird, der die angefragten Blöcke zwischenspeichert und bei einer erneuten Anfrage bereitstellt. Dabei kann bei jeder Übertragung Zeit eingespart werden, wodurch zum Beispiel das Hochfahren eines Gast-OS im Emulator signifikant beschleunigt wird.

Inhaltsverzeichnis

1 Einleitung

Emulation as a Service (EaaS) ist ein Dienst, mit dem archivierte Software emuliert werden kann. In der EaaS-Umgebung laufen Emulatoren, deren virtuelle Festplatten (Images) aus Image Archives geladen werden - von Servern, die Images über HTTP zur Verfügung stellen. Da die Emulatoren auf *Computing Nodes* in einem Rechnerverbund in der Cloud laufen und diese *Nodes* nach Bedarf spontan genutzt werden, befindet sich das Image Archive in der Regel nicht auf dem *Node* lokal, sondern auf einem zentralen Server, der die Images für mehrere *Nodes* bereitstellt. Der Zugriff erfolgt über HTTP, um die Anfragen an das Image Archive einfach steuern und filtern zu können.
Aktuell werden in der EaaS-Umgebung zur Emulation archivierter Systeme die gelesenen Festplatten-Blöcke der Images bei jeder Benutzung vom Image Archive geladen, da auf die Images ein wahlfreier Zugriff möglich ist und daher nicht das komplette Image vor dem Hochfahren des Gast-Betriebssystems geladen werden muss.
Solche Anfragen benötigen aber mehr Zeit als nötig, da wiederholt genutzte Blöcke auch mehrfach vom Image Archive geladen werden. Deshalb geht es darum, eine effiziente Lösung zum Zwischenspeichern bereits geladener Blöcke zu entwickeln. In diesem Szenario ist ein Zwischenspeicher dann effizient, wenn er verhindert, dass einmal geladene Blöcke erneut übertragen werden.
In der vorliegenden Arbeit werden die Anforderungen an einen Zwischenspeicher aufgestellt und in einem Lösungsvorschlag umgesetzt. Daraufhin wird die Umsetzung evaluiert und ein Ausblick auf mögliche Änderungen geboten.

2 Grundlagen

In diesem Kapitel werden die Problemstellung zur Zwischenspeicherung von Blockanfragen und die sich daraus ergebenden Anforderungen an einen Zwischenspeicher erläutert. Anschließend wird eine Lösung vorgeschlagen, die diese Anforderungen erfüllt.

2.1 Allgemeine Problemstellung

In der in Abbildung 2.1 dargestellten EaaS-Umgebung sind die Festplatten der emulierten Systeme (auch Gast-OS) als Festplattenabbilder (Images) über HTTP eingebunden. Die dadurch eigentlich unveränderlichen Festplatten - die HTTP-Verbindung ermöglicht dabei nur einen Lesezugriff - sind mit QEMU's[1] copy-on-write-Format (*qcow*) als *backing file* im Blocktreiber so eingebunden, dass sie beschreibbar sind. Sie sind selbst entweder im *raw*- oder ebenfalls im *qcow*-Format, wenn es sich um bereits veränderte Images handelt. Die Änderungen werden auf dem Host-OS zwischengespeichert und können am Ende einer Emulator-Sitzung zu einem neuen Image zusammengefasst

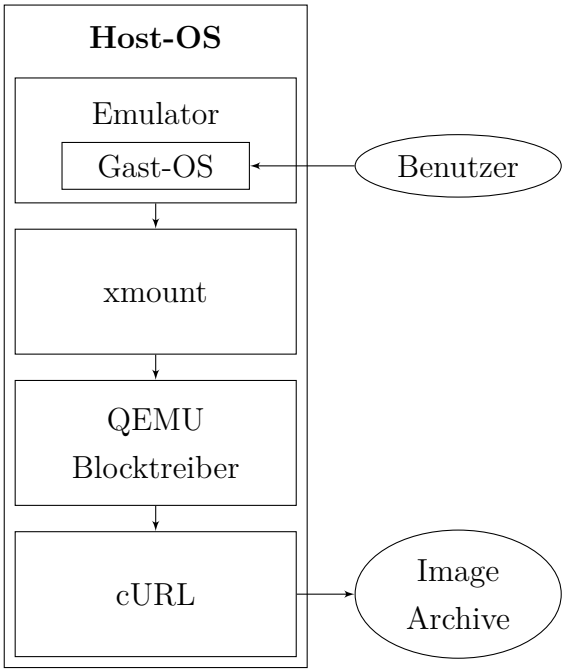

Abbildung 2.1:
Festplattenzugriff in der
EaaS-Umgebung

[1]engl. *Quick Emulator*, eine Virtualisierungssoftware zur Emulation von Hardware, erlaubt es, ein Gastsystem innerhalb des realen Host-Systems laufen zu lassen

werden - danach werden die Daten gelöscht. Die *xmount*[2]-Bibliothek ermöglicht es dabei, das als *qcow* vorliegende Image durch eine Konvertierung im vom Emulator benötigten Format (raw DD, VDI, VMDK,...) weiterzuverarbeiten.
Da sich diese Image Archives oft auf einem anderen Server - teilweise am anderen Ende der Welt - als der Emulator befinden, sind sie als sogenannte *backing files* eingebunden. Anstatt ein Image zu Beginn einer Sitzung komplett zu laden, ist ein wahlfreier Zugriff möglich. Dabei sorgt der QEMU Blocktreiber bei Images, die über HTTP eingebunden sind, dafür, dass die Daten zu Anfragen für Blöcke, die noch nicht geändert wurden und somit auch bisher noch nicht zwischengespeichert sind, beim *backing file* mit *cURL*[3] abgerufen werden. Dies wird so realisiert, dass jeder angefragte Block über einen HTTP-Range-Request[4] geladen wird. Ziel ist es, diese Range-Requests zwischenzuspeichern und dadurch das wiederholte Laden von einzelnen Blöcken zu verhindern.

[2]erlaubt es, spontan zwischen mehreren Input- und Output-Image-Formaten zu konvertieren [1]
[3]engl. *Client for URLs* [2], eine Bibliothek zur Übertragung von Daten
[4]Anfrage für eine Teildatei über HTTP, Genaueres im folgenden Abschnitt

2.2 Technische Problemstellung

Das Hypertext Transfer Protocol (HTTP) [3] wird genutzt, um Daten über Rechnernetze zu übertragen, in diesem Fall, um die Images einzubinden. Hier wird in Anfragen vom Client an den Server und Antworten vom Server an den Client unterschieden. Die unterschiedlichen Anfragemethoden können zum Beispiel genutzt werden, um Inhalte anzufordern (GET) oder Daten zu senden (POST).

Zum Senden von HTTP-Anfragen und -Antworten in der aktuell genutzten Version HTTP/1.1 wird eine TCP-Verbindung [4] benötigt. Beim Aufbau dieser Verbindung wird von beiden Seiten jeweils ein SYN-Paket gesendet und vom Gegenüber mit einem ACK-Paket bestätigt, zum Beenden der Verbindung läuft es mit FIN und ACK genauso. Wie in der Abbildung rechts erkennbar, können die SYN- beziehungsweise FIN-Pakete mit dem ACK-Paket zusammengefasst werden. Mit diesem Wissen kann man nun erkennen, dass für jeden Block, der vom Image Archive angefragt wird, mindestens drei Pakete hin und her geschickt werden. Die Zeit, die ein Paket zum Image Archive und zurück benötigt, wird *round trip time* genannt. Die Anfrage eines Blocks braucht also die doppelte *round trip time*, bis die Daten des Blocks angekommen sind. An den Abbildungen 2.3 und 2.4 kann man dies gut erkennen. Die Übertragungszeit liegt je nach Standort bei etwas mehr als dem Zweifachem der *round trip time*.

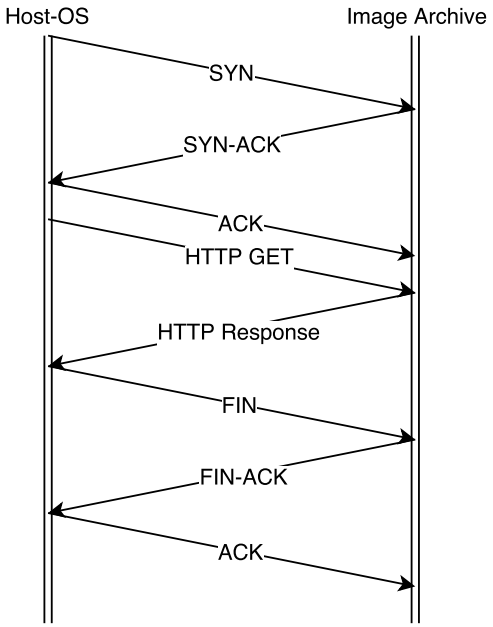

Abbildung 2.2:
HTTP-Verbindung über TCP

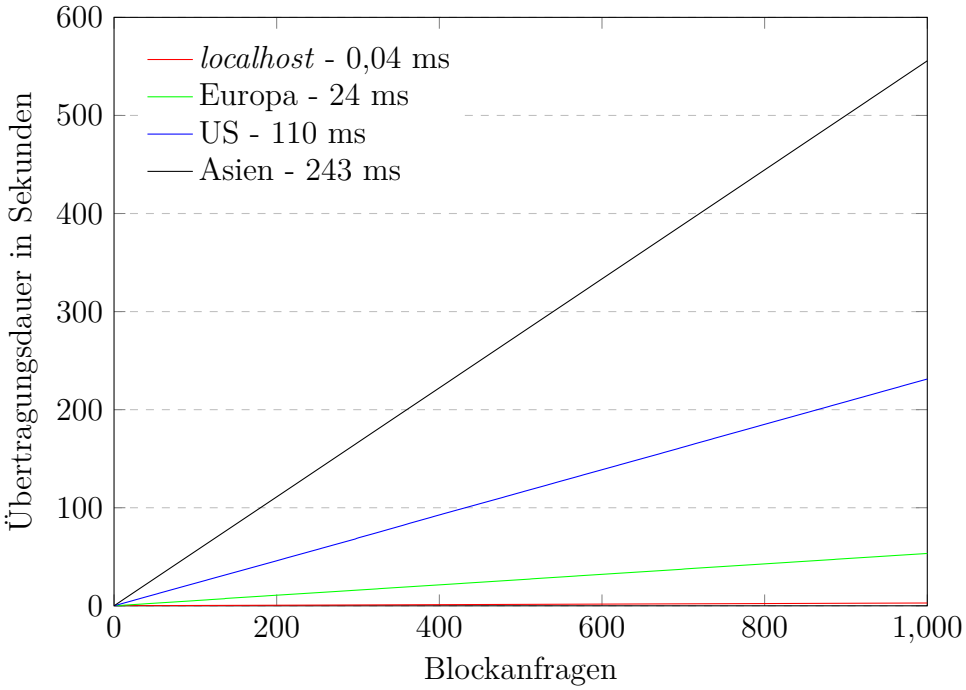

Abbildung 2.3: Gemessene Übertragungsdauer ohne Proxy

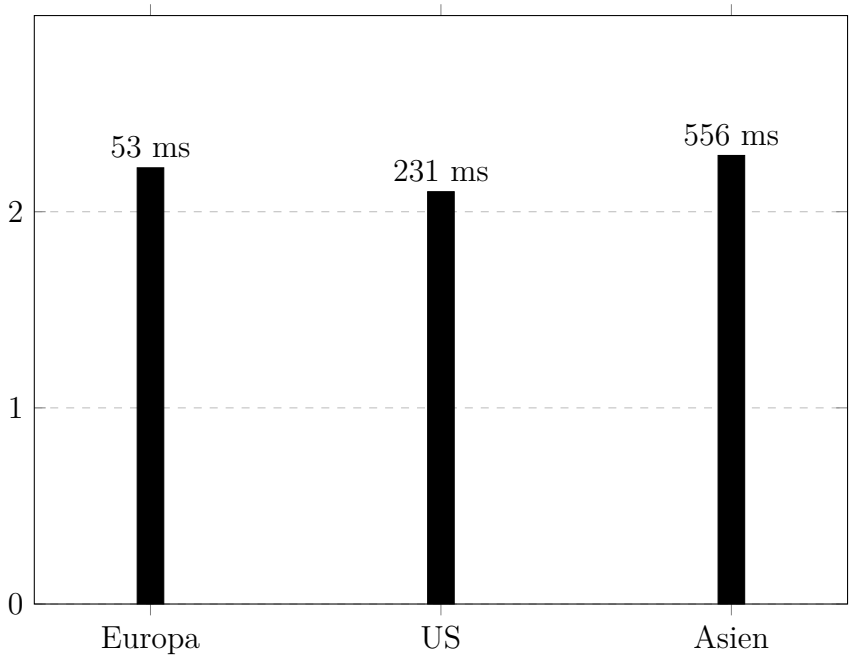

Abbildung 2.4: Durchschnittliche Übertragungsdauer ohne Proxy in Vielfachen der *round trip time*

HTTP-Nachrichten beinhalten sogenannte Header, die Informationen zur Anfrage bzw. Antwort enthalten und optionale Nutzdaten. Antworten haben außerdem einen Statuscode, der Informationen gibt, ob ein Fehler (und wenn ja welcher) vorlag oder die Anfrage erfolgreich beantwortet wurde. Ein solcher Header für GET-Anfragen ist „Range“ [5]. Hierdurch kann ausgedrückt werden, dass man nur einen spezifizierten Teil des angefragten Dokuments laden will. Dazu übergibt man das gewünschte Intervall und die genutzte Einheit. Eine erfolgreiche Antwort hat den Statuscode 206 statt des Codes 200 für die Bestätigung der Anfrage und beinhaltet als Nutzdaten den gewünschten Teil der Datei. Der Header „Content-Range“ bestätigt das Intervall und gibt die Größe der Datei in der gewählten Einheit an. Es ist auch möglich, mehrere Intervalle gleichzeitig anzufragen oder die Anfrage davon abhängig zu machen, ob oder wann sich die Datei geändert hat.
Das Laden von Festplattenblöcken aus einem Image Archive per Range-Request kann durch die vielen einzelnen Anfragen, für die jeweils ein Verbindungsaufbau nötig ist, so lange dauern, dass sie zum Beispiel das Hochfahren des Gast-Betriebssystems deutlich verzögert. Daher muss man einen Weg finden, die Fälle zu reduzieren, in denen Blöcke vom Image Archive geladen werden. Dies kann man erreichen, in dem einmal geladene Blöcke zwischengespeichert werden, so dass man sie bei einer erneuten Anfrage nicht nochmal vom Image Archive laden muss.
Die *qcow*-Schicht kann dazu nicht genutzt werden, weil sie einerseits bei einem Neustart neu erstellt wird. Außerdem wird die Implementierung durch die Tatsache erschwert, dass die geschriebenen Blöcke von den gelesenen getrennt werden müssen, um die ebenfalls in die *qcow*-Schicht geschriebenen, vom Benutzer vorgenommenen Veränderungen am Image speichern zu können. Da keine Bibliotheken verändert werden sollen, setzen wir zwischen *cURL* und dem Image Archive mit einem Proxy an.

Aus der Problemstellung kann man folgende Anforderungen an einen Proxy ableiten:

Umleitung der Anfragen zu einem Zwischenspeicher / Cache:
Um die Anwendung einfach zu halten, soll der Speicher für den Anwender möglichst transparent agieren. Das bedeutet, dass er ohne besondere Konfiguration von Seiten des Benutzers die Dateianfragen abfängt, zwischengespeicherte Blöcke zur Verfügung stellt und fehlende Blöcke nachlädt. Dazu müssen Blockanfragen an Image Archives zum Speicher umgeleitet werden. Anfragen, die keine zwischenspeicherbaren Range-Requests sind, müssen direkt weitergeleitet werden oder werden im besten Fall gar nicht umgeleitet.
Ein Problem bei der Umsetzung ist die Adressierung der unterschiedlichen Anfragen. Alle Anfragen kommen zunächst an den Proxy und dieser muss dann entscheiden, bei welchem Image Archive welches Image angefragt werden soll. Bei der Umsetzung sollten mehrere Image Archives gleichzeitig genutzt werden können. Also ist eine Unterscheidung der Anfragen zu den einzelnen Image Archives nötig.

Verwaltung des Zwischenspeichers:
Da die Images archiviert sind, verändern sie sich nicht mehr. Das heißt, dass einmal geladene Daten ohne Ablaufzeit zwischengespeichert werden könnten, auch über einen Emulatorneustart hinweg. Der einzige Grund für eine Löschung gespeicherter Dateien ist eine mögliche Beschränkung des hierfür zur Verfügung stehenden Speicherplatzes.
Bei einer erneuten Anfrage werden die bereits angefragten Blöcke vom Zwischenspeicher bereitgestellt, während fehlende Blöcke vom Image Archive nachgeladen werden. Wenn nur ein Teil der angefragten Blöcke zwischengespeichert wurde, entscheidet der Proxy, ob es effizienter ist, nur die fehlenden Blöcke zu laden oder die ganze Anfrage erneut vom Image Archive abzurufen. Falls mehrere Anfragen gesendet werden, um die fehlenden Blöcke zu laden, müssen die Verbindungen parallel aufgebaut werden, um keine Verzögerung durch den mehrfachen Verbindungsaufbau zu erhalten.

Geringer Wartungsaufwand:
Für den Zwischenspeicher sollten keine fremden Pakete wie *xmount* oder *cURL* umgeschrieben werden müssen, da sonst die Änderungen bei jedem Update eines Pakets in den aktualisierten Quellcode eingepflegt werden müssen.

Vorausschauendes Abrufen:
Optimal wäre, wenn der Speicher seine Leerlaufzeit nutzt, um vorausschauend Festplattenblöcke zu laden, bei denen es nach den letzten Anfragen wahrscheinlich ist, dass sie als nächstes abgerufen werden.

2.3 Lösungsansatz

Ein Proxy [6, Sektion 2.3] fungiert als Vermittler zwischen zwei Kommunikationspartnern, indem er Anfragen von einer Seite entgegennimmt und an die andere Seite weiterleitet. Dabei können Anfragen analysiert, gefiltert, verändert und umgeleitet werden. Proxys werden genutzt, um Funktionen wie Schutz, Protokollierung, Anonymisierung und ähnliches zu realisieren. Es wird unterschieden zwischen konventionellen Proxys, die vom Benutzer bewusst ausgewählt werden müssen, und transparenten Proxys, die zum Beispiel durch eine Umleitung im Router realisiert werden können. Diese sind für Benutzer nicht direkt sichtbar und benötigen daher auch keine besondere Konfiguration.
Die Blockanfragen (HTTP-Range-Requests) werden über einen Proxy geleitet, der auf der gleichen Maschine läuft wie der Emulator. Dieser Proxy speichert als Cache [7] die Antworten des Image Archives zwischen. Dadurch kann ein erneutes Laden vom Image Archive umgangen werden, ohne in die komplexe Emulation eingreifen zu müssen.
Der Proxy filtert aus den HTTP-Anfragen die Range-Requests heraus und überprüft, ob das angefragte Intervall im Zwischenspeicher vorhanden ist. Erst wenn der angefragte Block dort nicht vorhanden ist, wird er vom entfernten Server geladen und gespeichert, um ihn bei einer erneuten Anfrage anbieten zu können. HTTP-Anfragen, die keine Range-Requests sind, wer-

den an das Image Archive durchgeleitet und die Antwort dazu wird ohne Zwischenspeicherung zurückgegeben.
Um einen transparenten Proxy zu implementieren, werden die Domains der Image Archives zu eindeutigen nichtexistenten Domains umgeschrieben, bestehend aus der UUID[5] des Image Archives und der nicht vergebenen Top-Level-Domain *.invalid* [8]. Die Veränderungen müssen an den auf dem Host-OS und auch im Image Archive gespeicherten Adressen der Images vorgenommen werden. Im Image Archive sind diese nötig, da dort bereits veränderte Images liegen können, die wiederum selbst aus einer *qcow*-Schicht und dem Verweis auf ein weiteres Image bestehen. Im nächsten Schritt werden im Host-OS zusätzliche Informationen zur Adressauflösung gegeben, anhand derer Anfragen zu diesen Domains an den lokalen Proxy geleitet werden. Da der Port des Proxys eventuell abweicht, kann dieser noch mit *iptables* umgeleitet werden. Aus der Domain, an die die Anfrage zum Proxy kommt, kann dieser das Image Archive ableiten.
Der Proxy speichert erhaltene Blöcke für jede angefragte Ressource in sogenannten *sparse files*, die das komplette Image rekonstruieren, wobei allerdings noch nicht erhaltene Blöcke keinen Speicherplatz belegen. Um zu erkennen, welcher Block bereits geladen wurde, enthält eine weitere Datei zu jedem Block einen Binärwert, der diese Information bereitstellt.
Der Speicherplatz kann beschränkt werden, indem indem ein separater Thread den insgesamt belegten Speicher überprüft. Bei einer Überschreitung eines Limits werden die Cache-Dateien gelöscht, deren letzte Zugriffe am längsten her sind.
Um keine Probleme mit mehrfachem Zugriff auf die Cache-Dateien zu erhalten, wird ein *file locking* implementiert, das Dateien sperrt, die gerade bearbeitet werden und den Zugriff auf Dateien, die gelesen werden, nur weitere Leseoperationen erlaubt. Während einer etwaigen Bereinigung von Cache-Dateien aufgrund eines überschrittenen Speicherlimits wird der Thread des Proxys gesperrt, um nicht den seltenen Fehler zu haben, dass der Proxy die Datei mit den Binärdaten ausliest und erkennt, dass zwischengespeicherte

[5]engl. Universally Unique Identifier, ein eindeutiger Identifikator für *Nodes* in verteilten Systemen

Blöcke vorliegen und diese dann in der Zeit bis zum Zugriff auf das *sparse file* gelöscht wurden.
Gegenüber einem vorgefertigten Proxy hat eine eigene Lösung den Vorteil, dass sie weitere Möglichkeiten bietet: Zum einen kann die Verbindung zum Image Archive abgesichert werden, ohne im Proxy Probleme mit selbstsignierten Zertifikaten zu bekommen oder die Sicherheit anderweitig zu schwächen. Dazu wird zum Proxy eine unverschlüsselte Verbindung aufgebaut und dieser baut zum Image Archive eine verschlüsselte Verbindung auf. Um die Sicherheit der Übertragung so weit wie möglich zu verbessern und ein Mitlesen der übertragenen Daten zu verhindern, sollten alle Image Archives mit TLS verschlüsseltes HTTP unterstützen. Dadurch kann die Verbindung zwischen Proxy und Image Archive gesichert werden, ohne dass die Möglichkeit besteht, sie dadurch anzugreifen, dass man sich zwischen Proxy und Image Archive schaltet und behauptet, das Image Archive unterstütze keine verschlüsselte Übertragung. Außerdem kann eine Benutzerauthentifizierung an das Image Archive weitergegeben werden, indem ein Zugangsschlüssel (Token) in die angefragte Adresse codiert und vom Proxy weitergeleitet wird. Diese Möglichkeit wird in Abschnitt 4.1 genauer erörtert.

3 Evaluation

Um zu evaluieren, ob der Proxy die zum Laden eines Block benötigte Zeit verkürzt und wenn ja, wie groß diese Ersparnis ist, wird die Zeit gemessen, die Pakete zur Übertragung brauchen. Dabei werden jeweils 1000 Blöcke von je 512 Bytes angefragt, um auf einen verlässlicheren Wert zu kommen. Um die Relevanz der *round trip time* zu zeigen, werden die Messungen an drei Image Archives vorgenommen, die geographisch weit auseinander liegen und eine stark unterschiedliche *round trip time* haben. Dabei hat der Server mit Standort Europa eine *round trip time* von 24 ms, der Server an der US-amerikanischen Ostküste eine *round trip time* von 110 ms und der asiatische Server liegt bei 243 ms.

3.1 Szenarien

Die Messungen werden für unterschiedliche Szenarien durchgeführt. Je nach Szenario sind die übertragenen Blöcke entweder alle unterschiedlich, das heißt es gibt keine erneuten Übertragungen (*worst case*), oder sie sind alle die gleichen, so dass vom Proxy nur ein Block geladen werden muss (*best case*). Ein weiteres Szenario sind zufällig verteilte Blöcke, bei denen die Wahrscheinlichkeit auf eine erneute Übertragung mit jedem übertragenen Block steigt (*random case*), und zuletzt werden zum Vergleich die Blöcke ohne Proxy direkt vom Image Archive abgerufen. Da es dabei keinen Cache gibt, macht es hier keinen Unterschied, wie die Blöcke verteilt sind. Daher werden zufällig verteilte Blöcke verwendet.

3.2 Ergebnis

An Abbildung 3.1 kann man gut erkennen, dass der Proxy die *round trip time* eines Pakets im Vergleich zu einer direkten Übertragung leicht erhöht, wenn der Cache im *worst case* nicht zum Tragen kommt - durchschnittlich um 4 ms. Dieser Unterschied *(Overhead)* ist in der Abbildung blau markiert. Den *Overhead* benötigt der Proxy zur Verarbeitung der Anfrage des Gast-OS und Senden einer Anfrage an das Image Archive bzw. Laden des zwischengespeicherten Blocks aus einer Datei. Sie stimmt etwa mit der grün markierten Zeit überein, die der Proxy im *best case* ebenfalls zur Verarbeitung der Anfrage und dem Laden aus dem Cache benötigt - der Unterschied ist in der

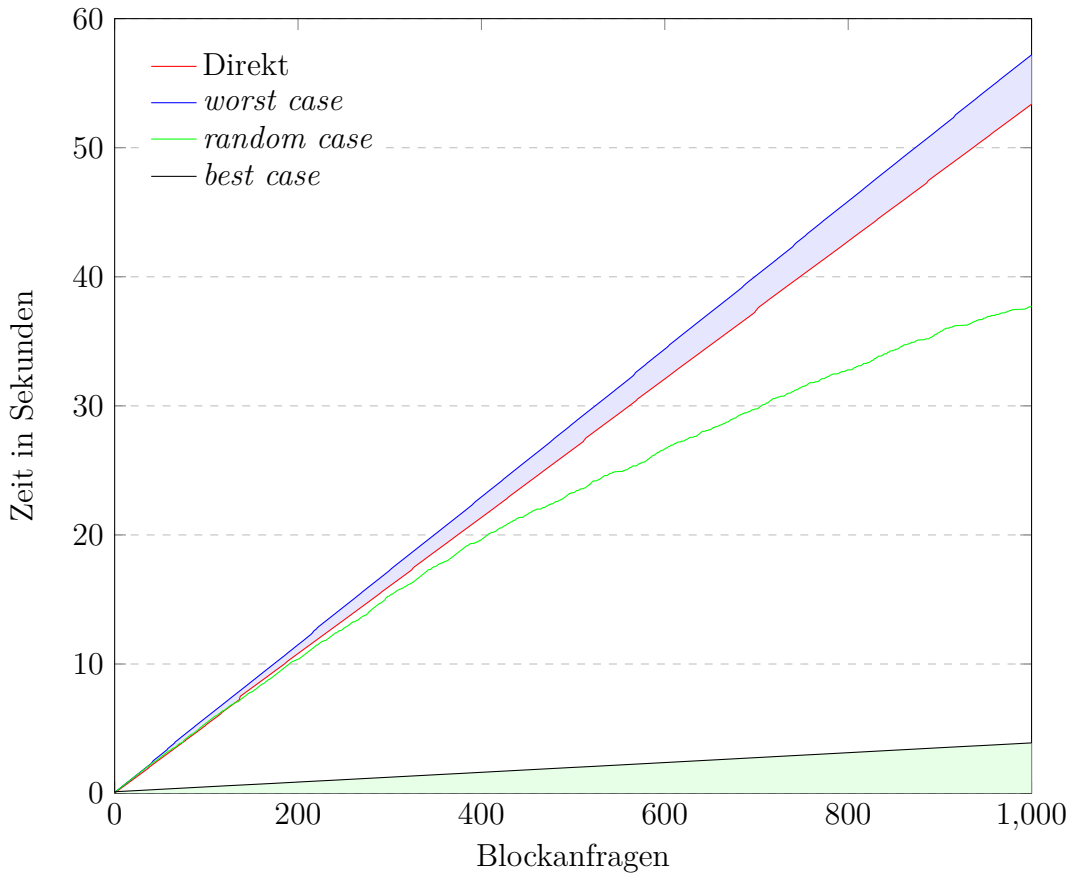

Abbildung 3.1: Übertragungsdauer Europa (24 ms)

Notwendigkeit begründet, den Block bei der ersten Anfrage vom Image Archive abzurufen und im möglichen Unterschied in der Dauer der Anfrage an das Image Archive zum Laden aus einer Datei. Trotz des *Overheads* bringt der Proxy schon bei niedriger *round trip time* einen Geschwindigkeitsvorteil, sobald Blöcke aus dem Cache geladen werden können, was man am *random case* erkennen kann. Wenn die *round trip time* sich wie in den Abbildungen 3.2 und 3.3 deutlich erhöht, weil das Image Archive außerhalb Europas in den USA beziehungsweise in Asien auf einem Server liegt, macht der *Overhead* im *worst case* nur noch einen verschwindend geringen Anteil aus und der Geschwindigkeitsvorteil beim Laden aus dem Cache kommt voll zum Tragen.

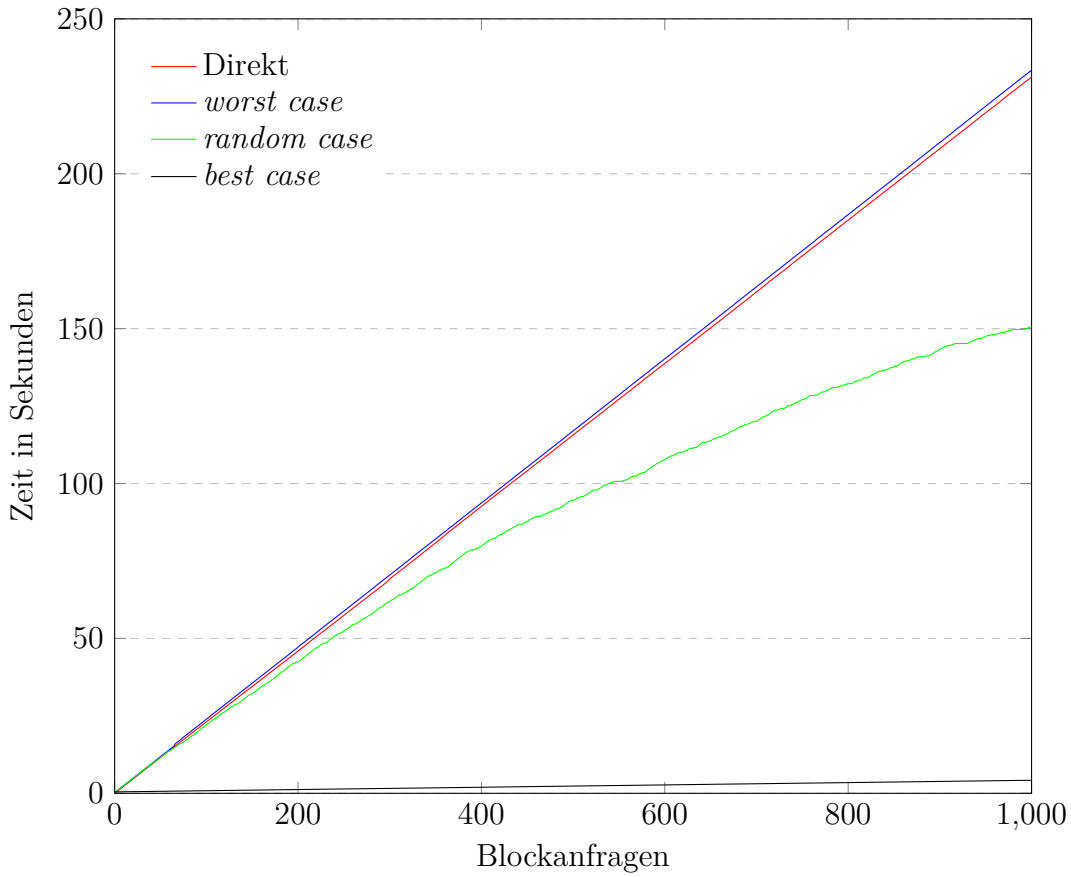

Abbildung 3.2: Übertragungsdauer US - Ostküste (110 ms)

Wie man in Abbildung 3.4 erkennen kann, benötigt eine direkte Anfrage des Emulators an das Image Archive etwas mehr als die zweifache *round trip time* - ähnlich wie in Abschnitt 2.2 beschrieben. Im *worst case* benötigt der Proxy wegen des *Overheads* etwas länger. Da dieser nicht von der *round trip time* abhängig ist, wirkt sich das auf die Übertragung zum europäischen Server im Vergleich zu dessen *round trip time* stärker aus. Diesen *Overhead* kann man auch im *best case* sehr gut erkennen, in dem die Auswirkung auf den europäischen Server ebenfalls hoch ist. Insgesamt verringert der Proxy die durchschnittliche Übertragungsdauer im *random* und *best case* auch relativ zur *round trip time* deutlich und macht die Übertragungsdauer bei vielen zwischengespeicherten Blöcken weniger abhängig von der *round trip time*.

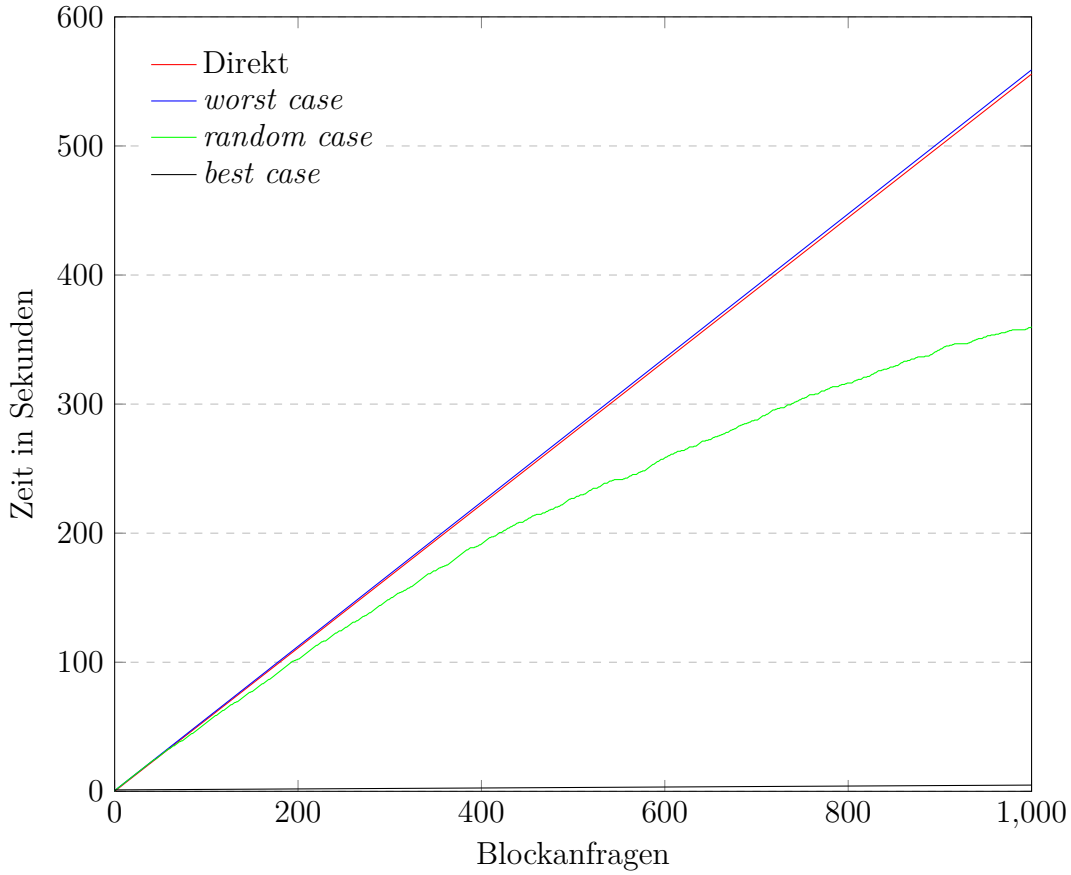

Abbildung 3.3: Übertragungsdauer Asien (243 ms)

In Abbildung 3.5 wird die Messung der Übertragungsdauer eines im Emulator laufenden Windows 98 mit Proxy dargestellt. Dabei wurden zum Hochfahren des Gast-OS 92 Megabyte, aufgeteilt auf etwa 340 Blöcke von durchschnittlich 277 Kilobyte Größe, übertragen. Davon waren 46 Megabytes in 76 Blöcken wiederholte Übertragungen. Nach einem Neustart des Betriebssystems war der Emulator bei allen Standorten etwa gleich schnell, da sich bereits alle benötigten Blöcke im Cache befanden. Im Verhältnis zur *round trip time* haben die Übertragungen sehr unterschiedlich lange gebraucht, waren aber trotzdem von dieser abhängig. Dies war beim Neustart dann nicht mehr der Fall, da keine Verbindung zum Image Archive mehr nötig war.

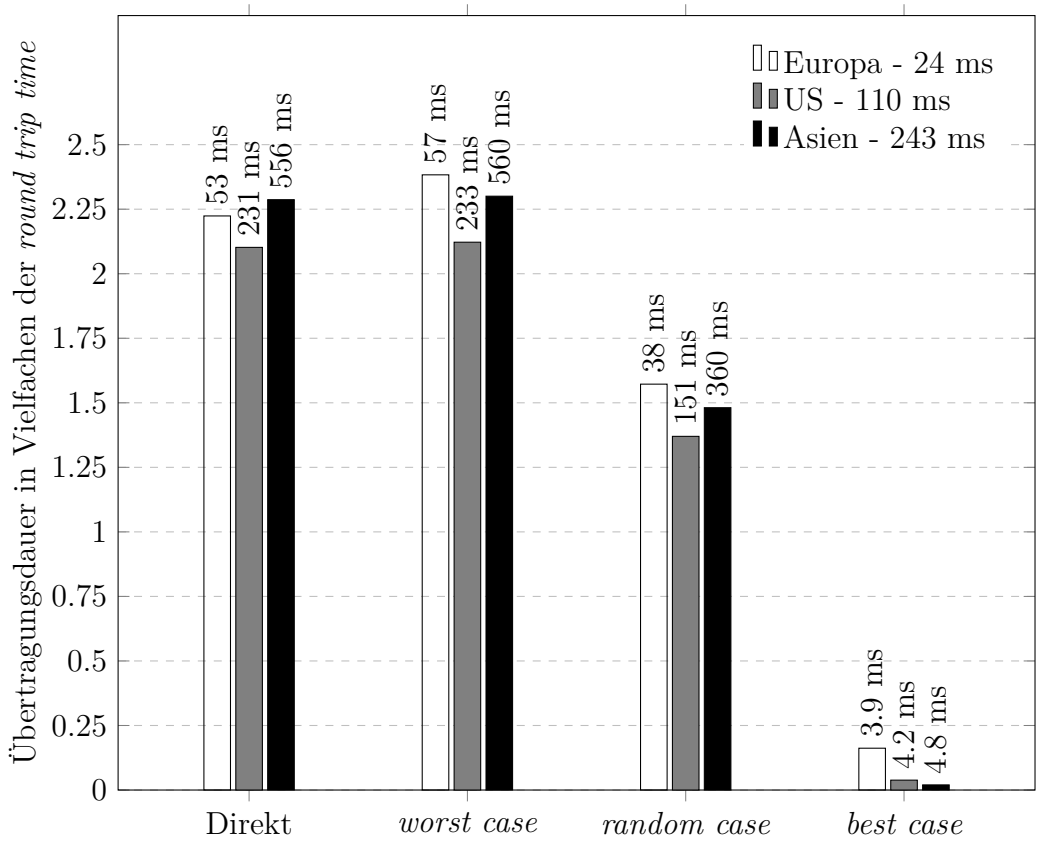

Abbildung 3.4: Durchschnittliche Übertragungsdauer

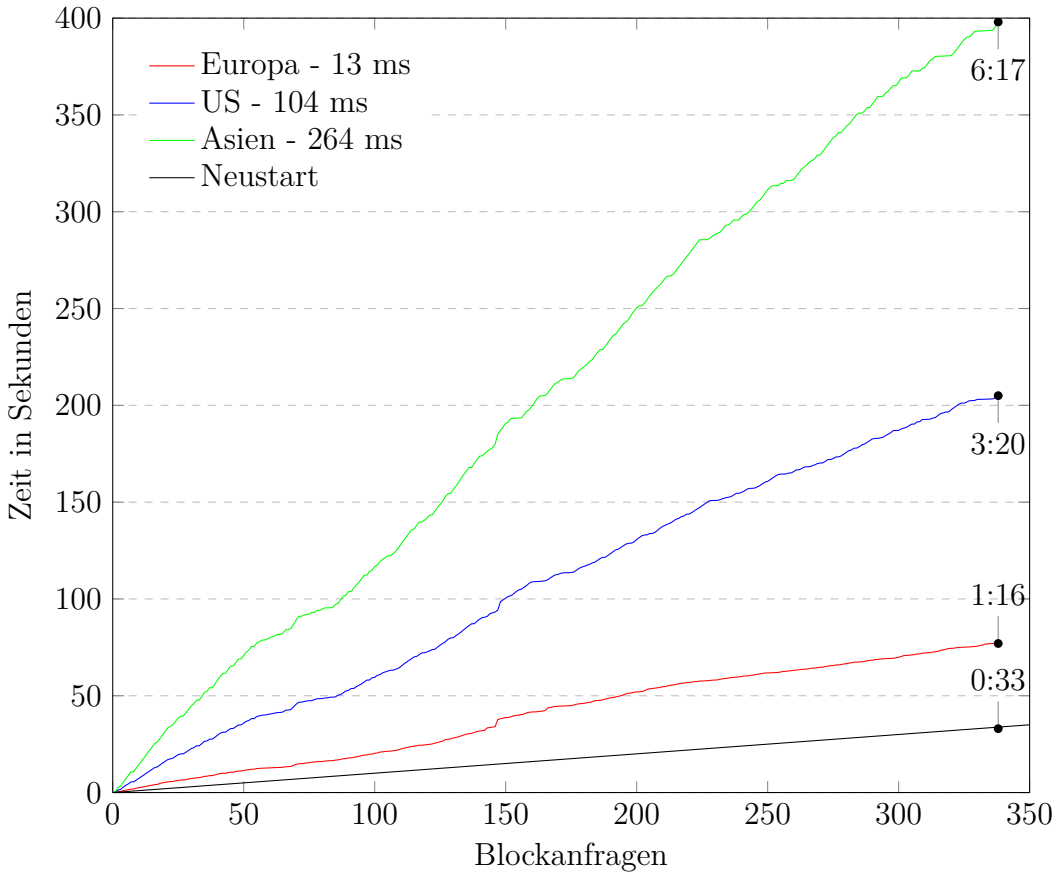

Abbildung 3.5: Übertragungsdauer mit Emulator

4 Ausblick

4.1 Authentifizierung

Die Image Archives unterstützen Benutzerauthentifizierung über zusätzliche HTTP-Header. Da diese allerdings im QEMU Blocktreiber nicht implementiert ist, kann sie nicht genutzt werden. Der Proxy kann aber einen in der angefragten Adresse - die einfacher modifiziert werden kann als der Blocktreiber - kodierten Zugangsschlüssel (*token*) auslesen und mit dem passenden Header mitschicken.

4.2 HTTP/2

Beim Wechsel auf HTTP/2 [9] verliert der Cache einen Teil seines Geschwindigkeitsvorteils durch die Kompression der HTTP-Header als Binärdaten und dadurch, dass bei Folge-Anfragen nur die veränderten Header mitgeschickt werden müssen. Zusätzlich werden bei HTTP/2 innerhalb einer Verbindung mehrere Streams geöffnet, die wiederum HTTP-Anfragen transportieren können. Außerdem wird nur noch eine TCP-Verbindung benötigt, daher gewinnt man gerade bei kleinen Anfragen durch das Weglassen des oftmaligen TCP-Verbindungsaufbaus Zeit. Nicht zuletzt können mehrere HTTP-Anfragen gleichzeitig nebeneinander laufen und müssen nicht auf den Abschluss der vorherigen Anfrage warten. Die letzten beiden Verbesserungen sind auch schon bei HTTP/1.1 mittels Pipelining [6, Sektion 6.3.2] möglich, werden aber vom Blocktreiber nicht unterstützt.
Andererseits kann HTTP/2 auch verwendet werden, um das Caching zu optimieren. In der Zeit, in der der Cache im Leerlauf ist, kann er vorausschauend schon weitere Teile des Images laden - dies wird *readahead* genannt. Durch die Priorisierung von HTTP/2 kann trotzdem schnell eine Anfrage vom Emulator geladen werden, während die weniger wichtigen präventiven Anfragen zurückgestellt werden.

5 Fazit

In dieser Arbeit wurden die Anforderungen an einen Zwischenspeicher für die EaaS-Umgebung aufgestellt und ein Lösungsansatz mithilfe eines Proxys vorgeschlagen. Dieser Proxy leitet die Blockanfragen mittels DNS-Einträgen um und erstellt aus den empfangenen Blöcken eine Datei, aus der er die Daten für wiederholte Anfragen lädt. Ohne den Proxy ist der Zeitaufwand für das Laden vieler Blöcke abhängig von der *round trip time* und daher bei entfernten Image Archives sehr hoch. Der vorgestellte Proxy kann bereits den Zeitaufwand für das Hochfahren eines Gast-OS senken, macht die Ladezeit bei einem Neustart unabhängig von der *round trip time* und senkt in den meisten Fällen die zum Übertragen mehrerer Blöcke benötigte Zeit. Keine Verbesserung bringt er, wenn keine Blöcke wiederholt angefragt werden, da dann der Cache nicht genutzt wird.
Über diese Arbeit hinaus sollte noch erarbeitet werden, wie die Erweiterungsmöglichkeiten genau umgesetzt werden können und welche Auswirkungen das mit sich bringt. Außerdem muss der Proxy noch vollständig in die bestehende EaaS-Umgebung integriert werden. Nicht zuletzt muss die in Abschnitt 2.3 vorgeschlagene Lösung zur Umleitung der Anfragen und richtigen Zuordnung der Image Archives umgesetzt werden. Bisher arbeitet der Proxy noch mit einem fest eingestellten Image Archive und die Anfragen vom Emulator müssen explizit umgeleitet werden.
In Zukunft kann der Proxy durch die Implementierung eines *readahead* noch beschleunigt werden und mit einer Benutzerauthentifizierung am Image Archive erweitert werden. Durch HTTP/2 kann das *readahead* mithilfe von Priorisierung verbessert werden. Ansonsten verliert der Proxy durch HTTP/2 etwas Zeitersparnis, was ihn aber nicht langsamer als die normale Übertragung ohne Zwischenspeicher macht.

Literatur

[1] Daniel Gillen. *XMOUNT | www.pinguin.lu*. Zugriff am 02.02.2018. URL: `https://pinguin.lu/xmount`.

[2] Daniel Stenberg. *curl - Frequently Asked Questions*. Zugriff am 02.02.2018. URL: `https://curl.haxx.se/docs/faq.html`.

[3] R. Fielding und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 7231. 2014.

[4] Information Sciences Institute University of Southern California. *Transmission Control Protocol*. RFC 793. 1981.

[5] R. Fielding, Y. Lafon und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Range Requests*. RFC 7233. 2014.

[6] R. Fielding und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing*. RFC 7230. 2014.

[7] R. Fielding, M. Nottingham und J. Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Caching*. RFC 7234. 2014.

[8] D. Eastlake und A. Panitz. *Reserved Top Level DNS Names*. RFC 2606. 199.

[9] M. Belshe und R. Peon. *Hypertext Transfer Protocol Version 2 (HTTP/2)*. RFC 7540. 2015.

Quellcode

```
from http import server, HTTPStatus
from hashlib import sha256
from configparser import ConfigParser
from datetime import datetime
from re import search
from urllib import request
from urllib.error import HTTPError
from io import StringIO
from contextlib import contextmanager
from heapq import heappush, heappop
from itertools import count
from time import sleep
import threading
import portalocker
import os
import sys

class CacheHandler(server.BaseHTTPRequestHandler):
    """ This class implements caching of Range-Requests.

    It subclasses the generic BaseHTTPRequestHandler and
    implements the HTTP GET, POST and HEAD methods.
    Other methods can be added easily if needed.

    """

    def do_GET(self):
        """ This method is called when handling a GET request.

        It filters range requests, forwards non-range requests and
        implements the caching by writing cached data to a sparse file
        and saving the information about which block is cached in another
        file.

        """
        rrange = self.headers.get("Range")
        if rrange is None or "," in rrange:
            try:
                headers = dict(self.headers.items())
                req = request.Request(target + self.path, None, headers)
                answer = request.urlopen(req)
                self.send_response(answer.getcode())
                for tup in answer.items():
```

```
                self.send_header(tup[0], tup[1])
            self.end_headers()
            self.wfile.write(answer.read())
        except HTTPError as e:
            self.send_error(e.code, e.msg)
    regex = search('([A-Za-z]+)=(\d+)\s*-\s*(\d+)', rrange)
    start = int(regex.group(2))
    start_sector = start//block_size
    end = int(regex.group(3))
    length = end-start+1
    length_sector = length//block_size
    m = sha256()
    m.update(self.path.encode('utf-8'))
    filename = "cache_" + m.hexdigest()
    data_filename = filename + ".data"
    missing_parts = 0
    threadLock.acquire()
    try:
        with self.filelock(filename, 'r') as f:
            f.seek(start_sector)
            cached_sectors = f.read(length_sector)
        last_exists = True
        cur = sector = missing_start = 0
        for cur, sector in enumerate(cached_sectors, start_sector):
            if sector is "1" and not last_exists:
                last_exists = True
                if missing_parts is 0:
                    missing_parts = (missing_start*block_size, (cur-1)*
                        block_size)
                else:
                    missing_parts = (start, start+length-1)
                    break
            elif not sector is "1" and last_exists:
                last_exists = False
                missing_start = cur
        if not last_exists:
            if missing_parts is 0:
                missing_parts = (missing_start*block_size, cur*
                    block_size)
            else:
                missing_parts = (start, start+length-1)
    except OSError:
        try:
            req = request.Request(target + self.path, None, {}, None,
                False, 'HEAD')
            size = int(request.urlopen(req).getheader("Content-Length"))
        except HTTPError as e:
            self.send_error(e.code, e.msg)
            return
        with self.filelock(filename, 'w') as f:
```

```
91                f.truncate(size//block_size)
92            with self.filelock(data_filename, 'wb') as df:
93                df.truncate(size)
94            missing_parts = (start, start+length-1)
95        part_start, part_end = missing_parts
96        try:
97            part_length = part_end - part_start + 1
98            rheaders = dict(self.headers.items())
99            rheaders["Range"] = "bytes=" + str(part_start) + "-" + str(
                  part_end)
100           req = request.Request(target + self.path, None, rheaders)
101           answer = request.urlopen(req)
102           data = answer.read()
103           if answer.getcode() == HTTPStatus.PARTIAL_CONTENT and len(data)
                  == part_length:
104               with self.filelock(data_filename, 'rb+') as f:
105                   f.seek(part_start)
106                   f.write(data)
107               with self.filelock(filename, 'r+') as f:
108                   f.seek(part_start//block_size)
109                   for i in range(part_length//block_size):
110                       f.write("1")
111               self.check_cache_size()
112           else:
113               raise HTTPError(
114                   req.get_full_url(),
115                   HTTPStatus.BAD_GATEWAY,
116                   "",
117                   rheaders,
118                   StringIO("")
119               )
120       except HTTPError as e:
121           self.send_error(e.code, e.msg)
122           return
123       with self.filelock(data_filename, 'rb') as f:
124           f.seek(start)
125           data = f.read(length)
126       threadLock.release()
127       size = os.stat(data_filename).st_size
128       self.send_response(HTTPStatus.PARTIAL_CONTENT)
129       self.send_header("Content-Length", length)
130       self.send_header("Content-Range", "bytes "+str(start)+"-"+str(end)+"
              /"+str(size))
131       self.end_headers()
132       self.wfile.write(data)
133
134
135   @contextmanager
136   def filelock(self, fp, mode = "r+"):
137       """ This method opens a file and locks it according to the mode.
```

```
138
139         The portalocker module is used to lock the requested file.
140         If a read only mode is chosen, a shared lock is used,
141         otherwise the lock is exclusive. The contextmanager closes
142         the file after leaving the "with" block and the lock is released
143         after closing automatically.
144
145         Args:
146             fp (str): The file path to open.
147             mode (str, optional): The mode used by open. Defaults to 'r+'
148
149         """
150         file = open(fp, mode)
151         if mode is 'r' or mode is 'rb':
152             flag = portalocker.LOCK_SH
153         else:
154             flag = portalocker.LOCK_EX
155         portalocker.lock(file, flag)
156         try:
157             yield file
158         finally:
159             file.close()
160
161
162     def do_POST(self):
163         """ This method is called when handling a POST request.
164
165         It forward the headers and data of the original request to the
166         image archive and returns the answer to the client.
167
168         """
169         try:
170             headers = dict(self.headers.items())
171             req = request.Request(target + self.path, self.rfile.read(),
                        headers, None, False, 'POST')
172             answer = request.urlopen(req)
173             self.send_response(answer.getcode())
174             for tup in answer.getheaders():
175                 self.send_header(tup[0], tup[1])
176             self.end_headers()
177             self.wfile.write(answer.read())
178         except HTTPError as e:
179             self.send_error(e.code, e.msg)
180
181
182     def do_HEAD(self):
183         """ This method is called when handling a HEAD request.
184
185         It forwards the headers of the original request to the
186         image archive and returns the answer to the client.
```

```
187
188         """
189         try:
190             headers = dict(self.headers.items())
191             req = request.Request(target + self.path, None, headers, None,
                    False, 'HEAD')
192             answer = request.urlopen(req)
193             self.send_response(answer.getcode())
194             for tup in answer.getheaders():
195                 self.send_header(tup[0], tup[1])
196             self.end_headers()
197             self.wfile.write(answer.read())
198         except HTTPError as e:
199             self.send_error(e.code, e.msg)
200
201
202 class QuotaCheck (threading.Thread):
203     """ This class checks for exceeding the allowed disk space and cleans
            update.
204
205     It subclasses the threading.Thread to run independently
206     from the proxy thread.
207
208     """
209
210     def __init__(self, quota):
211         """ This method is called to initiate the class.
212
213         It calls the super classes __init__ and sets the selected quota.
214
215         Args:
216             quota (int): The chosen disk usage limit.
217
218         """
219         threading.Thread.__init__(self)
220         self.quota = quota
221
222
223     def run(self):
224         """This method is called by the super class to run the thread.
225
226         It is called by the super classes start method.
227         It checks whether the space is exceeded. If this is the case,
228         it locks the proxys thread to avoid problems and cleans the cache.
229
230         """
231         while True:
232             sleep(1)
233             if self.cache_full():
234                 threadLock.acquire()
```

```
235                 self.clean_cache()
236                 threadLock.release()
237
238
239     def cache_full(self):
240         """ This method checks whether the disk space is exceeded. """
241         total_size = 0
242         for f in os.scandir():
243             if f.name.startswith("cache_") and f.is_file():
244                 total_size += f.stat().st_blocks/2048
245         return total_size > self.quota
246
247
248     def clean_cache(self):
249         """This methods reduces the disk usage to fulfill the quota
250
251         It creates a priority queue of cache files ordered by the time
252         of the last access and deletes the files whose last access
253         was longest ago until the quota is fulfilled.
254
255         """
256         total_size = 0
257         h = []
258         c = count()
259         for f in os.scandir():
260             if f.name.startswith("cache_") and f.is_file():
261                 total_size += f.stat().st_blocks/2048
262                 heappush(h, (f.stat().st_atime_ns, next(c), f))
263         while total_size > self.quota:
264             f = heappop(h)[2]
265             os.remove(f.path)
266             total_size -= f.stat().st_blocks/2048
267
268
269 class ServerThread (threading.Thread):
270     """ This class runs the proxy as a separate thread.
271
272     It subclasses the threading.Thread to run independently
273     from the quota thread.
274
275     """
276
277     def __init__(self):
278         """ This method is called to initiate the class.
279
280         It calls the super classes __init__.
281
282         """
283         threading.Thread.__init__(self)
284         self.httpd = server.HTTPServer(server_address, CacheHandler)
```

```
285
286
287     def run(self):
288         """This method is called by the super class to run the thread.
289
290         It is called by the super classes start method.
291         It runs the proxys method to check for incoming requests forever.
292
293         """
294         self.httpd.serve_forever()
295
296
297 if __name__ == '__main__':
298     """ This code starts the proxy and the quota threads
299
300     It is executed when the file is executed.
301     It reads the config file and sets the globals, creates the threads
302     and waits for the proxy thread to finish (which will not happen).
303     It catches a KeyboardInterrupt and ends the program if recieved.
304
305     """
306     config = ConfigParser()
307     config.read('config.ini')
308     server_address = ('', int(config['Config']['port']))
309     global block_size
310     block_size = int(config['Config']['block_size'])
311     global target
312     target = config['Config']['target']
313     global threadLock
314     threadLock = threading.Lock()
315     t1 = QuotaCheck(int(config['Config']['quota']))
316     t2 = ServerThread()
317     t1.start()
318     t2.start()
319     try:
320         while t2.isAlive():
321             t2.join(1)
322     except KeyboardInterrupt:
323         print("Interrupt recieved, stopping...")
324         sys.exit()
```