DISY
EIN DATEI-SYNCRHONISATIONSPROGRAMM

DANIEL M SWOBODA

Date: 2017-03-26.

Inhaltsverzeichnis

1. Lizenz

2. Aufgabenstellung

Aufgabe ist es ein Server-Client System zur Synchronisation von Dateien in Verzeichnissen über ein Netzwerk zu realisieren. Dabei sollen Zeitstempel und Hashes zum Dateivergleich genommen werden. Als Name für das Projekt wurde DiSy (Aussprache wie 'dizzy', engl. für 'schwindelig' da Verzeichnisse gleichgestellt werden, man sie quasi doppelt sieht), kurz für Directory-Sync, gewählt.

2.1. **Technologien.** Folgende Technologien wurden vorgegeben und/oder (mit Zustimmung von Prof. Kolousek) eingesetzt:

- C++14
- G++ 5.3 unter Ubuntu 16.04.2
- make und Cmake
- FMT
- SPDLOG
- JSON for Modern C++
- ASIO standalone
- C++17 Experimental Filesystem Library
- Google Protocol Buffers
- SHA256 Library von Zedwood.com

2.2. **Annahmen über die Umgebung.** Es wird davon ausgegangen, dass:

- die zwei Netzprozesse auf unterschiedlichen Geräten eingesetzt werden (Partner-IP muss vom User wählbar sein).
- die Verzeichnisse Unterverzeichnisse haben und diese mit-synchronisiert werden sollen.
- die Verzeichnisse vom User wählbar sein sollen.
- die Dateiberechtigungen übernommen werden sollen.
- das Löschen von Dateien nicht unterstützt werden muss.
- im Falle einer Unterbrechung der Verbindung ein inkonsistenter Zustand bis zur nächsten Synchronisation akzeptabel ist.
- Dateien mit gleichem Änderungsdatum synchronisiert sind.
- die Uhren auf beiden Geräten synchron sind.
- Es wird genau eine Synchronisation durchgeführt nach der sich sowohl Client als auch Server selbständig beenden.

3. Konzeption

DiSy ist als ein Server-Client System mit explizit getrennten Programmen für Server und Client konzipiert. Beide Programme nutzen großteils dieselben Funktionen welche in gemeinsam genutzten C++ Header Files organisiert sind.

3.1. **Klassenstruktur.** Neben den Hauptprogrammen gibt es die gemeinsam genutzten Header Files, die Klasse Config, die eingebunden Libraries und die durch protoc erzeugten Klassen für die Protocol Buffer Messages.

3.2. **Selbstangelegte Klassen und Header Files.** Zu den selbst angelegten Klassen und Header Files (abgebildet in Abbildung 1) gehören:

- Config (Klasse, Header-only): Laden, verwalten und bereitstellen von Konfigurationen.
- File (Library, Header-only): Alle Funktionen für den Datei-Zugriff.
- Message (Library, Header-only): Hilfsfunktionen für das erstellen, versenden und empfangen von Nachrichten.

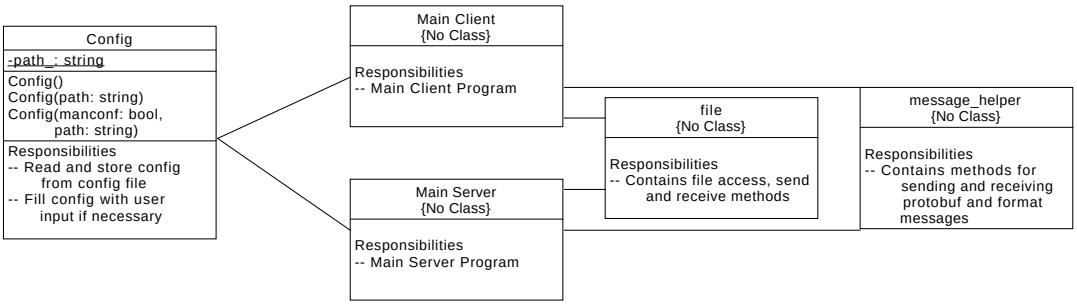

ABBILDUNG 1. Klassendiagramm welches den Aufbau und das Verhältnis der Dateien in DiSy darstellt.

3.3. **Programmablauf.** Der Programmablauf von DiSy sieht wie folgt aus:

- Laden der Konfiguration (evtl. Konfigurationsabfrage für Benutzer wenn Konfigurationsdatei nicht vorhanden)
- Verzeichnisindizierung
- Verbindungsaufbau
- Austausch der Synchronisierungszeit
- Austausch der Verzeichnisindizes
- Anfordern des Client Dateiindex
- Vergleich der Dateiindezes, bei gleichen Dateien Datumsvergleich
- Erstellung von fehlenden Verzeichnissen bei Server und Client erzeugen
- Anforderung der Hashes der Dateien die den gleichen Pfad aber unterschiedliche Änderungsdaten besitzen.
- Anfordern der Dateien die nur am Client existieren oder dort neuer und anders sind
- Senden der Dateien die nur am Server existieren oder dort neuer und anders sind
- Ändern der Änderungsdaten für alle Dateien um mehrfache und redundante Synchronisation zu vermeiden

3.4. **Konfiguration.** Die Konfiguration wird in einer JSON Datei gespeichert (für den Client: "client-c.json", für den Server: "server-c.json"). Diese wird, sofern nicht vorhanden, durch das Programm erstellt und mit Nutzereingabe von der Konsole befüllt.

4. Kommandozeilen Benutzerschnittstelle

Über die Kommandozeilen Schnittstelle lassen sich sowohl das Server als auch das Client Programm steuern.

4.1. **Argumente.** Server als auch Client besitzen die Argumente "-d" und "-p" mit denen fr die aktuelle Ausführung Verzeichnis und Port eingestellt werden können. Über "-h" lässt sich die Hilfe abrufen. Mithilfe von "-c" kann man die Konfigurationsdatei abändern".

4.2. **Aufbau.** Der Aufbau der Aufrufe sieht wie folgt aus:

- ./DiSy-Server [-h] [-c] [OPTION [VALUE]]...
- ./DiSy-Client [-h] IP [-c] [OPTION [VALUE]]...

5. Netzwerkkommunikation

Zur Netzwerkkommunikation wird die Bibliothek ASIO verwendet. Diese ist ein Wrapper um die Berkeley Sockets API von C++ und vereinfacht das erstellen von Programmen die Netzwerkverbindungen verwenden. DiSy setzt eine verbindungsorientierte und gesicherte Netzwerkverbindung voraus, aus diesem Grund wird auf Transportschicht TCP eingesetzt. Zur Kommunikation wird das DiSy-Protocol verwendet welches für diese Anwendung konzipiert wurde.

5.1. **Server.** Der Server nutzt die Klasse tcp::acceptor von ASIO um einen Listener auf dem eingestellten Port (standardmäßig 64446) zu erstellen. Sobald sich der Client verbindet wird ein tcp::socket Objekt erzeugt welches die Verbindung zum Client symbolisiert und über welches die Nachrichten gesendet werden. Nach dem Verbindungsaufbau wird der Handshake in Form von zwei Connect Nachrichten (die Erste von Client -> Server, die Zweite von Server -> Client) durchgeführt. Sobald die Synchronisation abgeschlossen ist werden Disconnect Nachrichten ausgetauscht (selbe Reihenfolge wie bei Connect) und die Verbindung abgebaut. Der Acceptor wird geschlossen und das Programm beendet.

5.2. **Client.** Im Gegensatz zum Server welcher eingehende Verbindungen akzeptieren muss, muss der Client nur eine Verbindung aufbauen können. Aus diesem Grund wird von Beginn an über ein tcp::socket Objekt gearbeitet. Mithilfe der Resolver-Klasse wird IP-Adresse+Port abgebildet und mit socket.connect() die Verbindung aufgebaut.

6. DiSy-Protocol

DiSy-Protocol ist ein zustandsbehaftetes Application-Layer Protokoll welches dem Austausch der Informationen und Dateien in DiSy dient. Es ist darauf ausgelegt zwischen genau einem Server und einem Client Informationen auszutauschen, Zustände sind nur für jeweils eine Verbindung gültig. Jegliche Datenübertragung findet in DiSy über die in diesem Protokoll definierten Nachrichten statt.

6.1. **Beschreibung.** Das DiSy-Protocol dient einerseits zum Austausch der Information über die zu Synchronisierenden Dateien, Dateihashes und der Dateien selbst. Dabei wird eine Mischung aus ASCII Nachrichten fester Größe und Serialisierten Protobuf Nachrichten beliebiger Größe übertragen.

6.2. **Aufbau der Nachrichten.** Eine DiSy Nachricht besteht aus einem 23-Byte langen ASCII Teil, welcher innerhalb des internen Sprachgebrauchs "Format Message"genannt wird, und einem n-Byte langen Binären Teil, welcher einen serialisierten Protobuf darstellt und zur Übertragung der Nutzdaten dient. Es wird immer eine Format Message geschickt bevor ein Datablock gesendet werden kann. Die "Datablock Message" ist optional und nicht für jede Nachricht notwendig und wird bei reinen Signal-Nachrichten weggelassen.

6.3. **Format Messages.** Diese Nachrichten bestehen aus einer 3 Zeichen langen Zeichenfolge welche den Nachrichtentyp identifizieren und einer 20 Zeichen langen ASCII Representation eines C++ size_t Datentyps. Der zweite Teil gibt die Länge (in Byte) an die der folgende Datenblock groß ist um ein korrektes Lesen zu gewährleisten. Da zur Kodierung ASCII (in 8-Bit Darstellung) verwendet wird entspricht die Länger der Zeichenkette der Länge der Nachricht in Byte.

Format Messages können so aussehen:

- ECN00000000000000000000
- SDR00000000000000000986

6.4. **Datablock Messages.** Datablock Messages sind über OStream und ASIO::-Streambuffer serialisierte und gesendete Protobuf Nachrichten diverser Typen. Während ein Nachrichtentyp eindeutig ist und nur in einem Zustand (dort auch mehrmals) eingesetzt werden kann, sind manche der Protobuf Nachrichten bei verschiedenen Nachrichtentypen eingesetzt da gleich-strukturierte Daten öfters gebraucht werden.

Protobuf dient im DiSy-Protocol dazu Nutzdaten einfacher und einheitlich zu serialisieren und den Zugriff auf die Daten einfacher zu gestalten.

6.5. **Protobuf-Messages.** Insgesamt werden sieben verschiedene Protobuf Messages in DiSy eingesetzt. Diese sind:

- Synctime: Beinhaltet einen int64-Block welcher genutzt wird um einen Unix-Timestamp zu speichern

TABELLE 1. Die Nachrichtentypen des DiSy Protokolls

Kurzer Name	**Langer Name**	**Richtung**	**Datenblock**
ECN	Connect	beide Richtungen	kein Datablock (C->S); Synctime (S–>C)
CDR	Create Directory	beide Richtungen	Dirlist
RDR	Request Directory	S->C	kein Datablock
SDR	Send Directory	C->S	Directory
RHS	Request Hash	S->C	Filelist
SHS	Send Hash	C->S	Hashlist
RFS	Request Files	S->C	Filelist
SFS	Send File	beide Richtungen	FileblockInfo
SFB	Send Fileblock	beide Richtungen	Fileblock
EFS	End Files	beide Richtungen	kein Datablock

- Directory: Beinhaltet eine Liste von Files Messages welches aus Pfad (string) und Änderungsdatum (int64) bestehen.
- Filelist: Beinhaltet eine Liste von Dateipfaden (string)
- Dirlist: Beinhaltet eine Liste von Verzeichnissen (string) und Berechtigungen (string) die erstellt werden sollen.
- Hashlist: Beinhaltet eine Liste von Dateipfaden (string), zugehörige SHA256 Hashes der Dateien (string) und Änderungsdatum (int64)
- FileblockInfo: Beinhaltet einen Dateipfad (string), die Anzahl der Fileblocks einer Datei (int64) und die Dateiberechtigungen (string).
- Fileblock: Beinhaltet einen Dateipfad (string) und einen Teil einer Datei (bytes).

Die Protobuf Messages Directory und Hashlist werden auch einseitig auf dem Server eingesetzt.

6.6. **Nachrichtentypen.** DiSy-Protocol kennt neun verschiedene Nachrichtentypen, diese sind in Tabelle 1 dargestellt.

6.7. **Protokoll Ablauf.** Der Ablauf des Protokolls ist in Abbildung 2, das Senden der Fileblocks in Abbiildung 3 dargestellt.

7. DATEIVERGLEICH

Um festzustellen welche Dateien vom Client zum Server, welche vom Server zum Client und welche garnicht übertragen werden sollen müssen die Dateiindizes des Servers und des Clients verglichen werden. Dies geschieht am Server.

7.1. **Dateilistenvergleich.** Zunächst werden alle Dateien im in der Konfiguration spezifizierten Ordner mithilfe der Filesystem Libary in einer Protobuf Directory Message erfasst und die des Clients an den Server übertragen. Daraufhin werden 3 Filelist Messages erzeugt. Zu einer werden alle Dateinamen hinzugefügt die nur im Clientindex auftauchen, in einer anderen werden alle Dateinamen hinzugefügt die

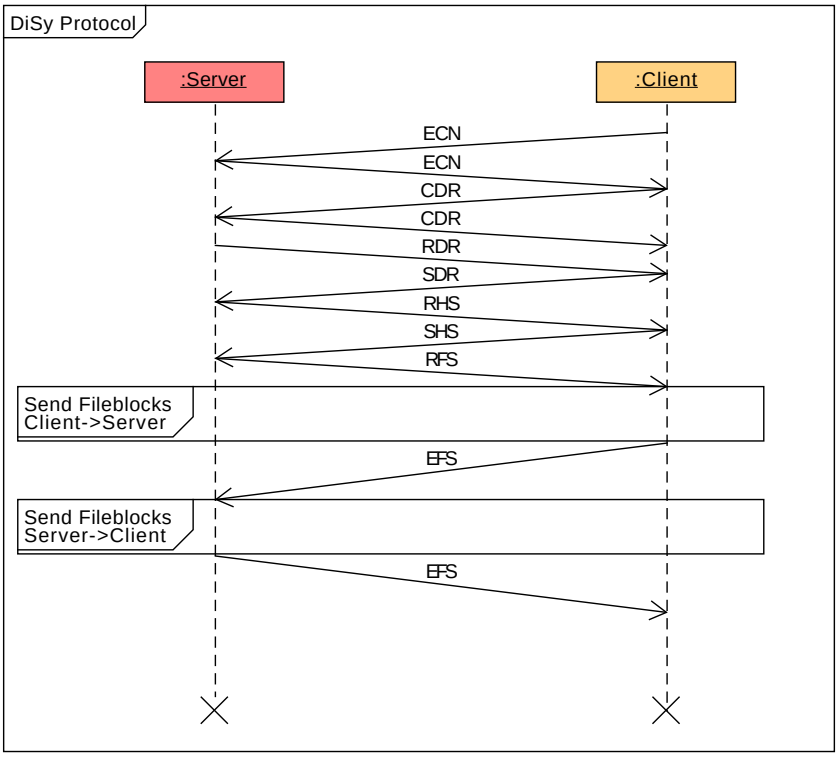

ABBILDUNG 2. Ablaufdiagramm, des zeitlichen Ablaufs des DiSy Protokolls.

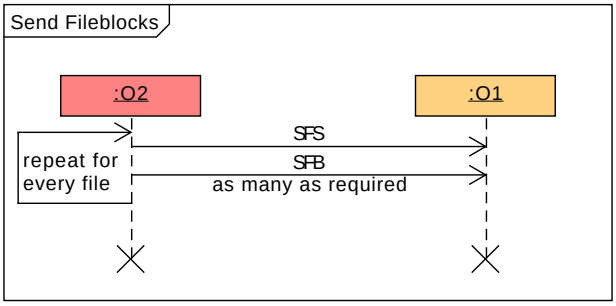

ABBILDUNG 3. Ablaufdiagramm des Sendevorganges des Fileblocks.

nur im Serverindex auftachen. Diese Beiden Filelists dienen dazu um die entsprechenden Dateien anzufordern. Im Gegensatz dazu dient die letzte für die Dateien die mit Hashvergleich verglichen werden müssen.

7.2. **Datumsvergleich.** Dateien die in beiden Indizes aufscheinen werden zunächst anhand ihres letzten Änderungsdatums verglichen. Ist dieses gleich wird davon ausgegangen, dass die Dateien bereits synchronisiert sind und diese werden von der Synchronisierung ausgenommen. Unterscheidet sich das Änderungsdatum muss ein Hashwertvergleich durchgeführt werden.

7.3. **Hashwertvergleich.** Alle Dateien die über Hashes verglichen werden, werden in eine Filelist Message gepackt und an den Client gesendet. Beide Geräte führen nun SHA256 Berechnungen auf die Dateien durch und speichern diese in Hashlist Messages. Der Client schickt seine Hashlist an den Server wo diese verglichen werden. Sind die Hashes unterschiedlich wird die Datei mit der neueren Änderungszeit bestimmt und in die entsprechende Filelist gepackt um diese später anzufordern. Sind die Hashwerte hingegen gleich werden die Dateien wieder ignoriert.

8. Dateiübertragung

Die Dateien werden in DiSy nicht als ganzes sondern in Teile geteilt übertragen. Dies ist notwendig damit beliebig große Dateien von DiSy gehandhabt werden können. Ein Teil des Codes ist in Listing 1 abgebildet.

Listing 1. Auschnitt des Codes zur Dateiübertragung

```
// send the sfs message
send_sfs(std::ref(sock), fbi);

char byte;
int count = 0;
//create filestram to read it
std::fstream file_strm(filepath.c_str(), std::fstream::in);
std::string datablock{};
//read in bytes
while (file_strm >> std::noskipws >> byte) {
    DiSyProto::Fileblock fb{};
    datablock += byte;
    count++;
    //send fileblock if blocksize is reached
    if(count == BLOCKSIZE) {
        fb.set_name(file.name());
        fb.set_data(datablock);
        send_fileblock(std::ref(sock), fb);
        //reset
        datablock.clear();
        count = 0;
    }
}
//send the last fileblock
DiSyProto::Fileblock fb{};
fb.set_name(file.name());
fb.set_data(datablock);
send_fileblock(std::ref(sock), fb);
```

8.1. **Einlesen.** Zum Einlesen der Dateien wird fstream genutzt. Es wird byteweise eingelesen und sobald die maximale Anzahl an Bytes oder das Ende der Datei erreicht wird in eine Fileblock Message geladen.

8.2. **Senden.** Beim Senden wird zunächst eine SFS Format Message gesendet. Diese beschreibt in ihrem Längenteil die Länge des darauffolgenden FileblockInfo Objekts. In diesem ist die Anzahl der Fileblocks enthalten, der Dateiname, sowie die Größe der normalen und des letzten Fileblocks. Diese Information vereinfacht das Schreiben. Die Fileblocks stellen eine Außnahme dar, da diesen keine eigene Formtat Message voran geht, sondern die Information der SFS für alle Fileblocks derselben Datei gilt. Gesendet werden die Fileblocks so wie jede andere Protobuf Message in DiSy.

8.3. **Empfangen.** Nachdem SFS und FileblockInfo gelesen wurden wird in einer Schleife die spezifizierte Anzahl an Fileblocks in Protobuf gelesen und deserialisiert. Ein Teil des Codes der zum Dateiempfang genutzt wird ist in Listing 2 abgebildet.

LISTING 2. Auschnitt des Codes zum Dateiempfangen

```
//receive the fileblock info
DiSyProto::FileblockInfo fbi{receive_fileblockInfo(std::ref(sock),
        extract_message_size(format))};

//print file to be receive
fmt::print("File: {}\n", fbi.name());
//if already existing, remove old file
try {
    fs::remove(path+fbi.name());
} catch(std::experimental::filesystem::v1::__cxx11::filesystem_error& e) {}

//open filestream to write to
std::ofstream filestrm;
filestrm.open(path+fbi.name(), std::ios::out | std::ios::app);

//for number of fileblocks receive a fileblock
for(int i = 0; i < fbi.number(); i++){
    DiSyProto::Fileblock fb;
    //get the SFB format message
    format = receive_format_message(std::ref(sock));
    //receive the fileblock with the given size
    fb = receive_fileblock(std::ref(sock), extract_message_size(format));
    //put the fileblock data into the filestream
    filestrm << fb.data();
}
//close the filestream after completion
filestrm.close();
//apply the file permissions
fs::permissions(path+fbi.name(), string_to_permissions(fbi.privileges()));
```

8.4. **Schreiben.** Die Fileblock Datenteile werden direkt über fstream in die entsprechend gleichnamige Datei im Zielverzeichnis geschrieben. Nach Abschluss des schreibens werden die Dateiberechtigungen der Ursprungsdatei auf die neu erstellte Datei gesetzt.

8.5. **Sonderfall: Verzeichnisse.** Verzeichnisse werden bereits im Vorhinein über die ausgetauschten Dirlists erstellt.

9. Sonderfälle und Problemfälle

Aufgrund des gegebenen Zeitrahmens und des Programmierers gibt es innerhalb von DiSy gewisse Sonder- und Problemfälle.

9.1. **Verhindern mehrfacher Synchronisation.** Da keine Informationen über die letzte Synchronisation gespeichert werden muss irgendwie anders eine redundante Synchronisation verhindert werden. Dies wurde, wie schon beschrieben, durch ein gleichstellen der Änderungsdaten erreicht, was nicht unbedingt die schönste Lösung ist.

9.2. **Übertragungsprobleme bei kurzen Zeitabständen.** ~~Bei zu kurzen Zeitabständen beim Senden von Fileblocks kann es zu Problemen kommen welche dafür sorgen das Dateien nicht richtig Empfangen werden.~~ Dieses Problem konnte behoben werden.

10. Performance

Für ein Programm wie DiSy ist es wichtig die Performance, insbesondere das Zeitverhalten, zu testen um Informationen über die Geschwindigkeit und die Effektivität zu sammeln.

10.1. **Testumgebung.** Als Testumgebung eine Virtuelle Maschine mit Ubuntu 16.04.2 eingesetzt, welche mit 3GB virtuellem RAM sowie 2 Kernen mit 4 Threads ausgestattet wurde. Als Virtualisierungssoftware wurde das properitäre VMWare Fusion in der Version 8.5.4 eingesetzt. Die virtuelle Maschine lief unter macOS 10.12.3 auf einem MacBook Pro (Retina, Mid 2012) mit einer 2,3GHz Intel Core i7 CPU, 8GB 1600MHz DDR3 RAM sowie NVIDIA GeForce GT650M (1GB Grafikspeicher) und Intel HD Graphics 4000 (1,5GB Grafikspeicher) Grafikkarten.

10.2. **Getestete Leistungsindikatoren.** Getestet wurden die Laufzeiten von einzelnen Operationen über interne Zeitmessungen im Programm sowie das Speicher und CPU Auslastungsverhalten im Vergleich zur Zeit während der Ausführung über ein externes Bash-Script und der Nutzung von PS.

Die Messgenauigkeit beträgt für Zeitmessungen 1ms sowie für CPU und Speicherverhalten 0.01% mit Aufzeichnungen alle 0.5s.

10.3. **Dateiübertragung.** In diesem Test wurde die Zeit zur Übertragung von Dateien ermittelt.

Zum Test der Dateiübertragungsgeschwindigkeit wurde ein mit 60 Binärdateien (PNG Bilddateien) gefüllter Ordner mit ca. 100MB Größe gewählt. Da Bildübertragung ein heutzutage häufig anzufindender Anwendungsfall von Dateisynchronisation ist (Google Photos, Dropbox Image Synchronisation, iCloud-Fotomediathek) ist wurde diese Art von Verzeichnis zum Testen gewählt.

Die Übertragungsgeschwindigkeit betrug im Durchschnitt über 4 Tests mit abwechselnden Übertragungsrichtungen 10,2315s.

10.4. **Dateiindizierung.** In diesem Test wurde die Zeit zur Erstellung des Dateiindex ermittelt.

Zum Test der Dateiindizierungsgeschwindigkeit wurde das durch cmake erstellte Verzeichnis "CMakeFiles" genommen da dieses über eine verzweigte Ordner-Struktur verfügt die dem eines Standard-Verzeichnisses ähnlich ist.

In der getesteten Version enthielt das Verzeichnis 51 Dateien. Die Indizierungsgewschindigkeit betrug im Durchschnitt über 4 Tests 1ms.

10.5. **Verzeichnisvergleich.** In diesem Test wurde die Zeit zum Vergleich der Dateiindizes auf dem Server ermittelt.

Zum Test der Vergleichsgeschwindigkeit wurde dasselbe Verzeichnis wie in Test 2 (Dateiindizierung) genutzt.

Im Durchschnitt über 4 Tests Betrug die Dauer des Verzeichnisvergleichs 0.25ms

10.6. **Hashwerterstellung und Vergleich.** In diesem Test wurde die Zeit zwischen RHS Nachricht und beendetem Hashwertvergleich gemessen. Die Erstellung der Hashes ist dabei inkludiert.

Zum Test wurde das gleiche Verzeichnis wie in Test 1 genommen, jedoch wurde eine Kopie mit anderen Änderungsdaten im Verzeichnis des Synchronisationspartners platziert damit von jeder Datei der Hashwert erstellt werden muss.

Im Durchschnitt über 4 Tests Betrug die Dauer der Hashwerterstellung und des Vergleichs 2,29s.

10.7. **Ressourcenauslastung während der Synchronisation.** In diesem Test wurde die vom Server verursachte Ressourcenauslastung gemessen und analysiert.

Zum Test wurde das gleiche Verzeichnis wie in Test 1 genommen und beim Server platziert, sodass die Nachrichten zum Client hin übertragen werden.

In Abbildung 4 wird das Testergebnis dargestellt. Während die RAM Auslastung konstant bei 0,1% bleibt steigt die CPU Auslastung mit dem Beginn der Übertragung nahezu Konstant auf fast 5% an.

11. Schlussfolgerung

DiSy ist ein funktionsfähiges System zum Übertragen von Dateien zwischen zwei Netzwerkprozessen unter dem Einsatz eines selbstentwickelten Protokolls, Google

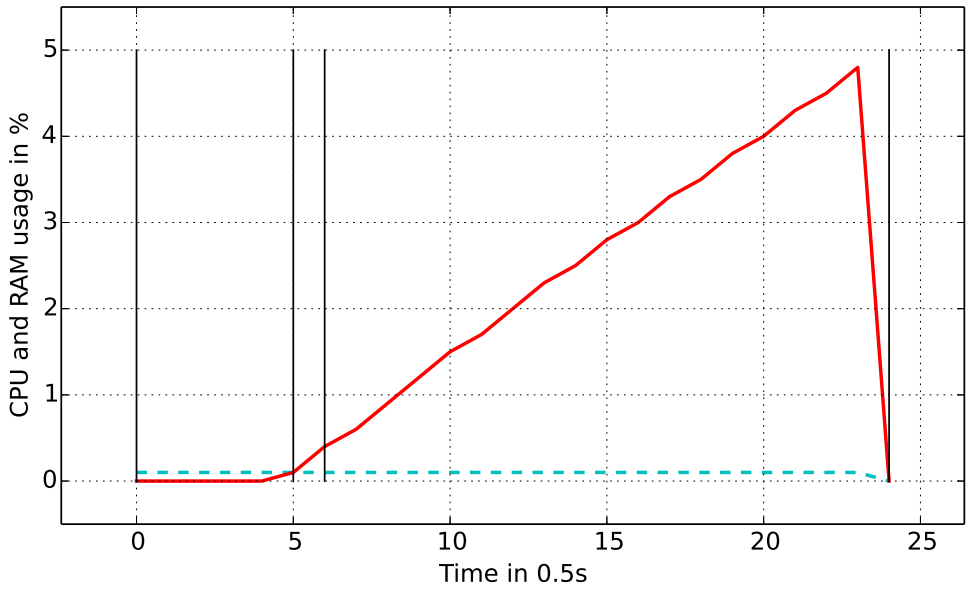

ABBILDUNG 4. CPU (rot) und RAM (blau, strichliert) Auslastung. Schwarze Striche markieren die Ereignisse: Programmstart, Client Verbindung, Dateien Senden beginn, Programm Ende in dieser Reihenfolge

Protocol Buffers und ASIO. Es ermöglicht eine Übertragung in angemessener Zeit und besitzt eingebaute Mechanismen zur Vermeidung von redundanten Synchronisationen sowie zur Vermeidung unnötiger Synchronisation durch die Anwendung von Zeitstempel und Hashverfahren. Dateinnamen, Verzeichnisstrukturen und Berechtigungen bleiben bei der Übertragung durch DiSy erhalten.

In Zukunft wäre eine Möglichkeit um Dateien zur Löschung zu Kennzeichnen denkbar.