2. NVS Projekt:
Verzeichnissynchronisation über Netzwerk

Name: Sebastian Grman
Klasse: 5BHIF
Katalognummer: 6
Beispielnummer: 42

Inhaltsverzeichnis

1 Einleitung

In diesem Projekt geht es um die Ausarbeitung einer Applikation zu Synchronisation von Verzeichnissen zwischen Netzwerkprozessen.

Ich gehe hier unter anderem auf den Synchronisationsalgorithmus basierend auf dem von *rsync*[4] und die Implementation der internen Kommunikation mit Pipes ein.

2 Algorithmus

Ein Algorithmus zur Synchronisation von Dateien über das Netzwerk, sollte idealerweise unnötige Datentransfers vermeiden. Er sollte erkennen, welche Teile der Dateien gleich geblieben sind, und nur die Teile, die sich verändert haben über das Netzwerk schicken. Andererseits soll auch vermieden werden, dass die an der Synchronisation beteiligten Geräte einer größeren Rechenbelastung als nötig ausgesetzt werden, und unter keinen Umständen sollte es vorkommen, dass eine Datei bzw. Teile einer Datei nicht synchronisiert werden, weil das Programm annimmt, dass sie bereits gleich sind.

2.1 rsync

Ein Synchronisations-Algorithmus, der diesen Anforderungen gerecht wird, ist der des Programms *rsync*[4]. *rsync* ist ein open source Programm zum inkrementellen Transfer von Dateien über das Netzwerk und ist besonders auf Netzwerke mit einer geringen Datenübertragungsrate ausgelegt[4].

Der Algorithmus, welchen *rsync* für den inkrementellen Datentransfer verwendet, wurde von Andrew Tridgell, einem der Erschaffer von *rsync*[5], in seiner Doktorarbeit beschrieben[6] und sieht wie folgt aus:

Der **rsync-Algorithmus** verwendet zwei unterschiedliche Signaturen, zur Überprüfung, ob zwei Datenblöcke gleich sind, eine schwache und eine starke[6].

Die schwache Signatur kann einfach und schnell berechnet werden und ist idealerweise eine rollende Signatur[1]. Wenn jetzt A mit B eine Datei synchronisieren will, so teilt A es dessen Version dieser Datei in Blöcke einer bestimmten Größe auf und sendet die schwachen Signaturen dieser Blöcke an B. B berechnet sich nun die schwachen Signaturen aller Blöcke dieser bestimmten Größe in dessen Version der Datei, das heißt, es wird beginnend bei jedem Byte ein entsprechend versetzter Block zur Berechnung herangezogen – deshalb ist es wichtig, dass die schwache Signatur eine rollende ist. B findet nun mithilfe einer Hashtabelle von seinen Blöcken die, die die gleiche schwache Signatur wie je einer der Blöcke von A hat. Für die Teile der Datei, bei denen mit der schwachen Signatur keinerlei Übereinstimmungen gefunden wurden, steht nun fest, dass sie

[1] Wenn man bei einer rollenden Signatur die Signatur eines Blocks hat, dann kann man, um die Signatur eines verschobenen Blocks zu bekommen, von der vorangehenden Signatur einen bestimmten Wert subtrahieren und einen anderen zu ihr addieren.

eindeutig unterschiedlich sind und ihr Inhalt muss transferiert werden. Für die laut der schwachen Signatur übereinstimmenden Blöcke berechnet B die starke Signatur.

Die starke Signatur sollte bei der benutzen Blockgröße eine möglichst geringe bzw. fast unmögliche Wahrscheinlichkeit haben, dass zwei unterschiedliche Blöcke – vor allem zwei nur leicht unterschiedliche Blöcke, da viele der stark unterschiedliche Blöcke schon von der schwachen Signatur unterschieden werden – die gleiche Signatur haben, dafür ist die Komplexität ein geringeres Problem, da ein Teil der Blöcke bereits ausgefiltert wurden. Für diesen Zweck eignen sich kryptographische Hashingfunktionen recht gut. Da diese Signaturen keine sicherheitstechnischen Bedeutungen haben, können auch Hashingfunktionen, die mittlerweile als kryptographisch unsicher gelten, benutzt werden. Andrew Tridgell schlug in seiner Arbeit *MD4* vor[6], inzwischen verwendet *rsync* aber schon *MD5*[4][5]. B schickt nun die starken Signaturen der Blöcke, bei denen die Übereinstimmung bzw. Nicht-Übereinstimmung noch ungewiss ist, an A. A berechnet nun die starken Signaturen der entsprechenden Blöcke dessen Version der Datei. Die Blöcke, bei denen die starken Signaturen nicht übereinstimmen, haben sich nun als unterschiedlich herausgestellt und ihr Inhalt muss transferiert werden. Die restlichen Blöcke, die Blöcke, bei denen sowohl die schwachen als auch die starken Signaturen unterschiedlich sind, werden nun als gleich angenommen.

Als letzter Schritt, zur Überprüfung, ob der Inhalt aller benötigter Blöcke transferiert wurde, werden die Gesamtsignaturen der Dateien miteinander überprüft. Wenn B die Datei von A haben wollte, so wird, nachdem B dessen neue Datei aus den Blöcken von A und dessen eigenen zusammengebaut hat, die Signatur von B's neuer Datei mit der Signatur von A's Datei verglichen. Sind die beiden Dateisignaturen gleich, so kann davon ausgegangen werden, dass der Transfer erfolgreich war und A und B jetzt die gleiche Datei haben. Sind die beiden Signaturen nicht gleich, so steht fest, dass der Transfer nicht erfolgreich war, und der ganze Algorithmus wird nochmal ausgeführt, aber mit einer neuen Signatur für jeden Block.

2.2 Benutze Abwandlung

Der Synchronisations-Algorithmus, welchen ich implementiert haben, ist nicht ident mit dem von *rsync*.

Aus Gründen der Einfachheit wurden einige Details des *rsync-Algorithmus*, welche durchaus bedeutend sind, von mir ausgelassen. So wird bei der Berechnung der Signaturen, statt einer Blockgröße, die die Datei in gleich große Blöcke unterteilt[6], immer eine statische Größe von 6.000 Byte genommen – diese Blockgröße gehört laut Tridgell zu den effizienteren, da man ein geringe Wahrscheinlichkeit von Überlappungen hat, es schlimmsten Falls aber bei den Signaturen, welche ich verwende, nur zu einem Overhead von knapp über 1% kommt (zwei 32 Byte große Signaturen). Der letzte Block – bzw. einzige, wenn die Datei kleiner als 6 KB ist – kann, sofern er nicht die genau Größe von 6 KB hat, daher nicht bei der Überprüfung der schwachen Signaturen nicht in die Hashtabelle

gegeben werden und auch nicht mit allen versetzten Blöcken verglichen werden. Stattdessen wird separat die Signatur des letzten Blocks gleicher Größe, lokal berechnet und verglichen. Weiters wird am Ende meines Algorithmus nicht die Gesamtsignatur der Dateien überprüft, dementsprechend wird der Algorithmus auch nie mit einer anderen Signatur wiederholt.

All diese Abwandlungen und fehlenden Details machen meinen Algorithmus selbstverständlich deutlich unzuverlässiger als den von *rsync*, doch wäre die Implementation sonst deutlich aufwendiger und eine zeitliche Fertigstellung des Programms sehr unwahrscheinlich gewesen.

Weiters ist zu vermerken, dass der Synchronisationsalgorithmus nur gestartet wird, wenn sich zwei Dateien mit dem selben Name bzw. selben Pfad vom Synchronisationsverzeichnis aus, beim Zeitstempel der letzten Änderung oder der Gesamtsignatur unterscheiden. Dieser Algorithmus ist nicht der Lage, Umbenennungen von Dateien zu erkennen, und wird stattdessen die Datei mit dem neuen Namen komplett über das Netzwerk transferieren und die Datei mit dem alten Namen löschen.

2.3 Schwache Signatur

Die rollende, schwache Signatur, welche in meinem Algorithmus benutzt wird, ist die selbe, die von Tridgell als erste beschrieben wurde[6]. Im Gegensatz zu einer gewöhnlichen Prüfsumme, in der alle Bytes einfach aufsummiert werden, bleibt diese Signaturfunktion nicht gleich, wenn vom Wert her das gleiche Byte zugerechnet wird wie abgezogen. Sie hat also ein geringere Kollisionswahrscheinlichkeit hat als eine gewöhnliche Prüfsumme und sieht wie folgt aus[6]:

$$r_1(k, L) = (\sum_{i=0}^{L-1} a_{i+k}) \bmod M \tag{1}$$

$$r_2(k, L) = (\sum_{i=0}^{L-1} (L - i)a_{i+k}) \bmod M \tag{2}$$

$$r(k, L) = r_1(k, L) + M \times r_2(k, L) \tag{3}$$

Nummer drei (3) zeigt die endgültige Formel für die Berechnung der schwachen Signatur, welche auf der *Adler-Prüfsumme* basiert[6]. $r(k, L)$ gibt die schwache Signatur mit der Versetzung k und der Länge L. a sind die Bytes der Datei und M kann ein beliebiger Wert zum Modulorechnen sein, in meinem Fall ist es 2^{16}. Das war jetzt die Formel für die *Neu*-Berechnung der Signatur, der für die Leistungsfähigkeit interessant Aspekt ist aber, dass man diese Signatur auch *rollend* berechnen kann, das heißt, um die Signatur eines Blocks zu bekommen, muss man nicht alle Bytes zusammenrechnen, sondern kann stattdessen auch die Signatur eines versetzten Blocks nehmen und muss mit diesem deutlich weniger rechnen. Die Funktionen für die rollende Berechnung der schwachen Signatur sehen wie folgt aus[6]:

$$r_1(k + 1, L) = (r_1(k, L) - a_k + a_{k+L}) \bmod M \tag{4}$$

$$r_2(k+1,L) = (r_2(k,L) - L \times a_k + r_1(k+1,L) \bmod M \tag{5}$$

$$r(k+1,L) = r_1(k+1,L) + M \times r_2(k+1,L) \tag{6}$$

Diese rollende Eigenschaft der Signatur bringt große Leistungsvorteile bei der Berechnung der schwachen Signaturen aller Blöcke bei allen Versetzungen.

2.4 Starke Signatur

Als starke Signatur wird, wie dies auch bei *rsync* der Fall ist, *MD5* verwendet[4]. Diese Hashingfunktion sollte eine genügend niedrige Kollisionswahrscheinlichkeit haben.

3 Implementierung

Die Implementierung dieser Applikation erfolgte wie gefordert in *C++*. Es handelt sich hierbei nicht nur um die Implementierung des bereits beschriebenen Synchronisationsalgorithmus. Es musste auch die Netzwerkkommunikation zwischen den Programmen und die interne Kommunikation zwischen den einzelnen Komponenten der Applikation realisiert werden.

3.1 Algorithmus

Der Algorithmus wurde implementiert, so dass er wie bereits beschrieben funktioniert. Der Großteil der Implementierung ist in der Datei *src/file_operator/sync_system.cpp* zu finden, wobei Teile des Algorithmus, wie etwa die Signaturfunktionen, sich in anderen Modulen befinden.

Die **schwache Signatur** wurde einfach aus ihrer bereits beschriebenen mathematische Darstellung in *C++*-Code umgeschrieben.

Für die **starke Signatur**, also *MD5*, wurde *OpenSSL* verwendet, welches diese Funktion leicht zu verwenden macht[2].

Auf die restliche Implementierung des Algorithmus hier nun konkret einzugehen würde aufgrund der Komplexität und Durchwachsenheit dieser Teile des Programms den Rahmen sprengen.

3.2 Netzwerkkommunikation und Synchronisationsverlauf

Die Kommunikation zwischen einem Synchronisations-Client und einem Synchronisations-Server inklusive Verlauf des Synchronisationsprozesses sieht wie folgt aus:

1. Der Client initialisiert die Kommunikation, indem er eine Anfrage an den Server schickt, ihm eine Liste sämtlicher seiner synchronisierbaren Dateien zu schicken.

2. Der Server geht dieser Anfrage nach und schickt dem Client ein Liste sämtlicher Dateien.

3. Der Client vergleicht nun diese Liste mit seinen lokalen Dateien. Dateien, die er nicht kennt, werden vom Server angefordert, Dateien, von denen er weiß, dass sie bei ihm bereits gelöscht sind, werden dem Server entsprechen als zu löschen vermittelt. Bei Lokale Dateien, die nicht in der List des Servers aufscheinen, schickt der Client eine Anfrage, ob diese gelöscht wurden oder der Server sie vom Client braucht. All restlich Dateien, die, die mit ihrem Namen bzw. Pfad sowohl beim Client als auch Server vorhanden sind, werden auf Übereinstimmung bezüglich Zeitstempel und Gesamtsignatur überprüft. Für die Dateipaare, bei denen diese Wert unterschiedlich sind, wird der Synchronisationsalgorithmus initialisiert und der Client schickt eine entsprechende Synchronisationsanfrage mit den schwachen Signaturen an den Server.

4. Der Server folgt den Nachrichten des Clients und sendet entsprechende Antworten zurück (z.B. eine angeforderte Datei). Bekommt der Server eine Synchronisationsanfrage so überprüft er die schwachen Signaturen mit der seiner lokalen Datei. Gibt es Übereinstimmungen, so schickt der Server ein Anfrage für die starken Signaturen der übereinstimmenden Blöcke an den Client. Abhängig von den Zeitstempeln der Dateien schickt der Server für die nicht übereinstimmenden Blöcke entweder seine Inhalte an den Client oder fragt die entsprechenden Inhalte beim Client an. (Der mit dem kleineren Zeitstempel, sprich der älteren Datei, muss sich die Daten holen, um den neueren Stand der Datei rekonstruieren zu können.)

5. Der Client schickt die geforderten starken Signaturen an den Server und, wenn gefordert, auch den Inhalt der nicht übereinstimmenden Blöcke bzw. speichert der Client die erhaltenen Korrekturdaten zur späteren Dateikonstruktion.

6. Der Server überprüft die starken Signaturen den Clients mit seinen eigenen. Wie bei den schwachen auch, schickt er für die nicht übereinstimmenden Blöcke die Korrekturdaten an den Client bzw. fordert sie an. Mit den übereinstimmenden Blöcken wird nichts mehr gemacht.

7. Der jenige, der die Korrekturdaten erhalten hat, (Client oder Server) muss nun die neue Datei rekonstruieren. Dafür wird aus Sicherheitsgründen eine temporäre Datei im Unterverzeichnis *.sync* erstellt. Diese wird nun in der entsprechenden Reihenfolge nacheinander mit den Korrekturdaten befüllt und für die Stellen, für die es keine Korrekturdaten gibt, wird der Inhalt aus der alten, lokalen Datei verwendet, da diese an dieser Stelle gleich sein muss, zumindest laut den Signaturen, sonst gebe es ja Korrekturdaten dafür. Am Ende wird die zusammengebaute Datei an ihren richtigen Ort verschoben und die alte somit verschoben.

Die Netzwerkkommunikation ist mithilfe von *Protocol Buffers*[3] für die Serialisierung der Nachrichten und *Asio*[1] zur Übertragung der serialisierten Nachrichten realisiert.

Da *Asio* eine Bibliothek zur textbasierten Netzwerkübertragung ist und *Protocol Buffers* die Nachrichten in ein binäres Format serialisiert, kann es dazu kommen, dass in den Nachrichten das Zeichen zur Trennung von Nachrichten und dem beenden des Leseflusses für *Asio* vorkommt. In diesem Fall ist es das *Newline*-Zeichen, `\n`. Da *Protocol Buffers* dieses Zeichen im Serialisierungsformat durchaus benutzt, werden die serialisierten Nachrichten noch zusätzlich in Base 64 kodiert und müssen dementsprechend am anderen Ende wieder dekodiert werden, noch bevor sie deserialisiert werden können. Dadurch werden Problem zwischen der textbasierten Netzwerkkommunikation und der Serialisierung vermieden.

3.3 Interne Kommunikation

Die Applikation besteht aus mehreren Prozessen, die voneinander halbwegs unabhängig agieren sollten. Es sind drei Haupt-Prozesse, die für die Funktionstüchtigkeit der Applikation verantwortlich sind:

- Der **Server** ist für die Kommunikation mit anderen Clients zuständig und startet für jede Verbindung einen eigenen Thread.
- Der **Client** ist für die Kommunikation mit einem Server zuständig.
- Der **File Operator** ist für den gesamten Synchronisationsablauf und der Kommunikation mit dem Dateisystem zuständig, sowohl Client- als auch Server-seitig.

Neben diesen drei Hauptprozessen gibt es noch einen Prozess, welcher sich um **Kommandozeile** kümmert.

Die Kommunikation zwischen diesen Akteuren erfolgt über **Pipes**. Der File Operator bekommt das empfangende Ende einer Pipe und der Server und der Client (auch die Kommandozeile) bekommen das sendende Ende dieser Pipe, über welches sie Anfragen an den File Operator schicken können, welche dieser bearbeiten soll. Damit der File Operator ihnen Antworten auf ihre Anfragen zurückschicken kann, schicken sie das sendende Ende einer Pipe mit, welche sich in ihrem Besitz befindet. Dieses kann der File Operator benutzen, um ihnen Rückmeldungen zu schicken.

Um die Unterscheidung von sendenden und empfangenden Enden von Pipes zu ermöglichen, wurden Interfaces (rein abstrakte Klassen) und Klassenvererbung herangezogen. Bei den sendenden und empfangenden Enden einer Pipe handelt sich um zwei Interfaces, welche von der Pipe implementiert werden.

Wie aus dem Klassendiagramm in Abbildung 1 ersichtlich ist, haben das sendende Ende, `SendingPipe`, und das empfangende Ende, `ReceivingPipe`, eine Menge aus gemeinsamen Operationen, welche im Interface `Closable` definiert sind, ein Pipe ist nämlich schließbar. Was aus dem Diagramm auch noch ersichtlich ist, ist dass die beiden Enden der Pipe und so auch die Pipe generell generisch für einen beliebigen Typ sind. Soll ein Prozess mit einer Pipe jetzt nur

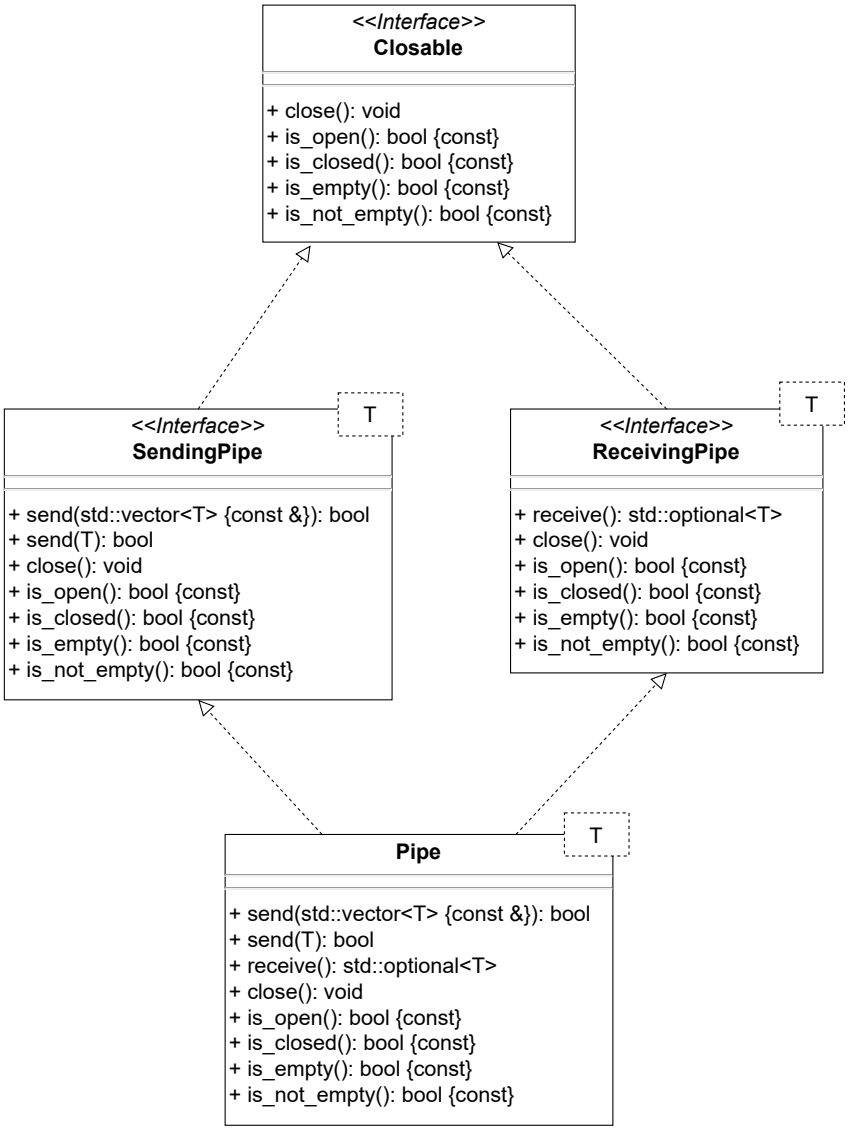

Abbildung 1: UML-Klassendiagramm von Pipe

senden bzw. nur empfangen können, so bekommt dieser nur die Schnittstelle `SendingPipe` bzw. `ReceivingPipe` zur Verfügung gestellt.

Schauen wir uns nun die Implementierung von `Pipe` etwas genauer an, diese muss schließlich auch threadsafe sein.

```
1  std::lock_guard pipe_lck{pipe_mtx};
2  if (is_open()) {
3      msgs.push(std::move(msg));
4      receiving_finishable.notify_one();
5
6      return true;
7  }
8  else {
9      return false;
10 }
```

Listing 1: Senden einer Nachricht ... bool send(T msg)

Hie sehen wir zuerst die Funktion zum senden, `bool send(T msg)`. Da es sich um eine schließbare Pipe handelt muss überprüft werden, ob die Pipe überhaupt noch offen ist. Von Interesse ist, dass das Mutex für die Pipe bereits vor der Überprüfung, ob die Pipe offen ist, also noch bevor überhaupt entschieden wurde, ob gesendet werden kann, gesperrt wird. Dies hat damit zu tun, dass die Funktion `close()`, zum Schließen der Pipe, auch dieses Mutex sperren muss. Durch das frühzeitige Sperren des Mutex beim Senden wird vermieden, dass die Pipe nach der Abfrage, ob sie offen ist, aber noch vor dem eigentlichen Senden, geschlossen wird. Das gleiche sieht man auch beim Empfangen. Ansonsten passiert nicht viel mehr außer, dass die Nachricht eben in die Queue, welche der interne Speicher der Pipe ist, gelegt wird und anschließend wird einer der aufs Empfangen Wartenden benachrichtigt. Wenn das Senden erfolgreich war, sprich die Pipe offen war gibt es `true` zurück, ansonsten `false`. Neben dieser Methode gibt es noch `bool send(std::vector<T>& msgs)` zum Senden von Nachrichten. Diese funktioniert sehr ähnlich, kann jedoch mehrere Nachrichten nehmen und legt sie alle gleichzeitig in der erhaltenen Reihenfolge in die Queue, sobald dies möglich ist.

```
std::unique_lock pipe_lck{pipe_mtx};
if (is_open()) {
    // warten, dass es etwas zum Empfangen gibt
    // oder die Pipe geschlossen wurde
    receiving_finishable.wait(
        pipe_lck,
        [this](){
            return is_not_empty() || is_closed();
        }
    );

    if (is_open()) {
        T msg{std::move(msgs.front())};
        msgs.pop();

        return msg;
    }
    else {
        return std::nullopt;
    }
}
else {
    return std::nullopt;
}
```

Listing 2: Empfangen einer Nachricht ... std::optional<T> receive()

Beim Empfangen mit `std::optional<T> receive()` haben wir, wie bereits erwähnt, auch den Fall das als aller erstes die Mutex gesperrt wird und dann wird wieder überprüft, ob die Pipe offen ist. Ist die Pipe offen, wird mit der Condition Variable `receiving_finishable` überprüft, ob es etwas zum Empfangen gibt und, falls nötig, darauf gewartet. Ist das Warten vollendet wird nochmals überprüft, ob die Pipe offen ist, da das Warten auch durch das Schließen der Pipe beendet wird. Wenn ja, holt sich die Methode die vorderste Nachricht aus der Queue, löscht sie aus der Queue raus und gibt sie an den Aufrufer der Methode zurück. Konnte keine Nachricht geholt werden, da die Pipe geschlossen ist, wird Nulloption zurückgegeben. Mit einem optionalen Wert als Rückgabewert wird im Programm deutlich ausgedrückt, dass das Empfangen einer Nachricht nicht unbedingt erfolgreich sein muss.

```
std::lock_guard pipe_lck{pipe_mtx};
open = false;
receiving_finishable.notify_all();
```

Listing 3: Schließen der Pipe ... void close()

Das Schließen der Pipe mit `void close()` ist deswegen interessant, weil es auch das Mutex der Pipe sperrt, obwohl man auf den ersten Blick meinen könnte, dass dies unnötig ist: Niemand schreibt auf die Variable `open` mit `true` und früher oder später würden alle sehen, dass die Pipe geschlossen ist. Der Grund, dass close() das Mutex sperren muss, liegt bei der Condition Variable. receive() verwendet die Condition Variable `receiving_finishable` zur Überprüfung sowohl auf empfangsbereite Nachrichten, als auch auf das Schließen der Pipe, denn in beiden Fällen will man nicht mehr warten. Würde close() das Mutex nicht sperren, bevor es mit der Condition Variable alle Wartenden benachrichtigt, könnte es passieren, dass ein Prozess gerade zufällig überprüft, ob er mit dem Warten aufhören kann, weil das Mutex isst ja nicht gesperrt, also kann er das, während close() die Benachrichtigung an alle schickt. In diesem Fall würde dieser eine Prozess die Benachrichtigung verpassen und so im wartenden Status verbleiben. Aus diesem Grund muss close() auch das Mutex sperren.

Damit haben wir uns nun einen kurzen Überblick über ein paar wichtige und interessante Teile des Programms verschafft.

4 Bedienung

Es gibt viele verschiedene Möglichkeiten die Applikation zu bedienen. Parameter kann mittels CLI, Umgebungsvariablen oder einer Konfigurationsdatei in *JSON* bestimmen. Weiters kann man, sobald das Programm gestartet ist, über eine Kommandozeile noch zusätzlich mit dem System interagieren. Die genauen Details zu all diesen Interaktionsmöglichkeiten lassen sich in der *README* nachlesen. Hier wird nun die grundsätzliche Bedienung überblicksmäßig beschrieben.

Zur Synchronisation eines Verzeichnisses muss man über die Konsole in dieses Verzeichnis navigieren und von dort aus das Programm `sync` starten. Das Programm übernimmt beim Starten das derzeitige Arbeitsverzeichnis als Synchronisationsverzeichnis. Beim Starten kann man festlegen, ob man einen Synchronisations-Server, einen Synchronisations-Client oder beides haben will. Um einen Server zu starten reicht das Kommandozeilenargument `-s`, dies startet einen Server der auf allen Eingangsadressen auf Port 9876 hört. Um einen Client zu starten, muss man das Kommandozeilenargument `-a` und eine Adresse oder Hostnamen angeben, der Client verbindet sich dann zu diesem Host auf Port 9876 und startet die Synchronisation. Um beide als ein Systemprozess zu starten, muss man beide Kommandozeilenargumente angeben. Der Server läuft solange bis man ihn nicht abbricht und der Client macht nach dem die Synchronisation vollendet ist nichts mehr, muss allerdings auch manuell abgebrochen werden.

Beim Synchronisationsverzeichnis sollte sicher gegangen werden, dass das Programm die entsprechenden Rechte hat, um alle Dateien zu lesen und zu beschreiben. Im Synchronisationsverzeichnis wird außerdem noch das Unterverzeichnis *.sync* erstellt, in welches man lieber keine Dateien speichern sollte, da dieses Unterverzeichnis die Datenbankdatei für die Applikation enthält und

sämtliche temporären Dateien zum Aufbau der neueren Versionen auch hier erstellt und beschrieben werden. Außerdem werden Dateien im Unterverzeichnis *.sync* nicht synchronisiert.

Literatur

[1] *Asio*. URL: https://think-async.com/Asio/ (besucht am 11. 04. 2021).

[2] *OpenSSL, MD5*. URL: https://www.openssl.org/docs/man1.1.1/man3/MD5.html (besucht am 11. 04. 2021).

[3] *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers (besucht am 11. 04. 2021).

[4] *rsync*. URL: https://rsync.samba.org/ (besucht am 11. 04. 2021).

[5] *rsync - Wikipedia*. URL: https://en.wikipedia.org/wiki/Rsync (besucht am 11. 04. 2021).

[6] Andrew Tridgell. „Efficient Algorithms for Sorting and Synchronization“. Feb. 1999, S. 49–69. URL: https://www.samba.org/~tridge/phd_thesis.pdf.