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 in diesem Dokument auf den Synchronisationsalgorithmus, welcher auf dem von *rsync*[5] basiert und durch die Benutzung von zwei Signaturen eine möglichst geringe Netzwerkbelastung ohne übermäßige Rechenanforderungen erzielt, ein. Weiters wird auch auf die Implementierung der Netzwerkkommunikation mittels *Asio*[1] und *Protocol Buffers*[4], den Synchronisationsablauf und die Implementierung der internen Kommunikation zwischen den einzelnen Akteuren eingegangen. Als letztes wird die Bedienung des Programms beschrieben.

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, welche 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 diese bereits gleich sind.

2.1 rsync

Ein Algorithmus, der diesen Anforderungen gerecht wird, ist der des Programms *rsync*[5]. *rsync* ist ein open source Programm zum inkrementellen Transfer von Dateien über das Netzwerk und ist besonders auf Netzwerke mit einer geringen Datenübertragungsrate ausgelegt[5]. Der Algorithmus, welchen *rsync* für den inkrementellen Datentransfer verwendet, wurde von Andrew Tridgell, einem der Erschaffer von *rsync*[6], in seiner Doktorarbeit beschrieben[7] und sieht wie folgt aus:

Der **rsync-Algorithmus** verwendet zwei unterschiedliche Signaturen, zur Überprüfung, ob zwei Datenblöcke gleich sind, eine schwache und eine starke[7].

Die **schwache Signatur** kann einfach und schnell berechnet werden und ist idealerweise eine rollende Signatur – die schwache Signatur und das Prinzip der rollenden Signaturen wird in Abschnitt 2.3 genauer beschrieben. Wenn jetzt A mit B eine Datei synchronisieren will, so teilt A 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, welche die gleiche schwa-

che Signatur wie je einer der Blöcke von A haben. Für die Teile der Datei, bei denen mit der schwachen Signatur keinerlei Übereinstimmungen gefunden wurden, steht nun fest, dass sie 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 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 wurde. Für diesen Zweck eignen sich kryptographische Hashingfunktionen recht gut. Da diese Signaturen keine sicherheitstechnischen Bedeutungen in der Applikation haben, können auch Hashingfunktionen, die mittlerweile als kryptographisch unsicher gelten, benutzt werden. Andrew Tridgell schlug in seiner Arbeit *MD4* vor[7], inzwischen verwendet *rsync* aber schon *MD5*[5][6]. 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 auch 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. Das Gerät, welches die neuere Datei haben will und die Inhalte der ungleichen Blöcke empfangen hat, kann nun die Datei lokal zusammenbauen. Für die als als gleich angenommenen Blöcke werden die Inhalte der lokalen Datei herangezogen und die fehlenden Daten wurden bereits transferiert.

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 wiederholt, aber mit einer neuen Signatur für jeden Block[7].

Dieser Algorithmus schafft es Dateien über ein Netzwerk zu synchronisieren, ohne das Netzwerk und die beteiligten Geräte stark in Anspruch zu nehmen, und stellt damit eine ausgezeichnete Basis für meinen Algorithmus dar.

2.2 Benutze Abwandlung

Der Synchronisations-Algorithmus, welchen ich implementiert haben, ist nicht ident mit dem von *rsync*, da dieser eine in Teilen eine recht komplexe Implementierung erweisen würde, welche sowohl meine fachlichen als auch die zeitlichen

Einschränkungen dieses Projekts sprengen würde. Also werden aus Gründen der Einfachheit einige Details des *rsync-Algorithmus*, welche durchaus bedeutend sind, ausgelassen.

Bei der Berechnung der Signaturen wird, statt einer Blockgröße, welche die Datei in gleich große Blöcke unterteilt[7], immer eine statische Größe von 6.000 Byte genommen – diese Blockgröße gehört laut Tridgell zu den effizienteren, da man ein geringe Wahrscheinlichkeit von Kollisionen hat und es schlimmsten Falls bei den Signaturen, welche ich verwende, nur zu einem Overhead von deutlich unter 1% kommt (zwei Signaturen der insgesamten Größe von 36 Byte – siehe die Abschnitte 2.3 und 2.4 – relative zu einem 6 KB großen Block). Der letzte Block – bzw. einzige, wenn die Datei kleiner als 6 KB ist – kann, sofern er nicht die genau Größe von 6 KB aufweist, daher 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 zeitige Fertigstellung des Programms sehr unwahrscheinlich.

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

Bei der schwachen Signatur ist es, wie bereits erwähnt, von großer Bedeutung für die Leistungsfähigkeit des Programms, dass diese eine rollende Signatur ist.

2.3.1 Rollende Signatur

Eine rollende Signatur ist eine Signatur, welche einerseits klassisch mit einer Funktion, welche die entsprechenden Daten, von welchen man die Signatur haben will, nimmt und sie zusammenrechnet, berechnet werden kann, andererseits aber auch inkrementell auf Basis einer bereits bekannten Signatur, wodurch sich große Rechenersparnisse ergeben können. Das heißt: Wird die Signatur eines Blocks an der Stelle k mit der Funktion $r(k)$ berechnet, so kann die Signatur des um eins versetzen Blocks an der Stelle $k+1$ mit der Funktion $r(k+1)$ auf Basis der bereits bekannten Signatur des Blocks an der Stelle k und den Unterschieden zwischen den beiden Blöcken berechnet werden.

Die rollende, schwache Signatur, welche in meinem Algorithmus benutzt wird, ist die selbe, welche von Tridgell als erste beschrieben wird[7]. 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 wie abgezogen wird. Sie hat also ein geringere Kollisionswahrscheinlichkeit und sieht wie folgt aus[7]:

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

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

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

Die Gleichung 3 zeigt die endgültige Formel für die Berechnung der schwachen Signatur, welche auf der *Adler-Prüfsumme* basiert[7]. $r(k, L)$ 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, dass man diese Signatur auch rollend berechnen kann und die Funktionen für die rollende Berechnung der schwachen Signatur sehen wie folgt aus[7]:

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

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

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

Diese rollende Eigenschaft der Signatur bringt große Leistungsvorteile vor allem bei der Berechnung der schwachen Signaturen aller Blöcke mit allen Versetzungen. In der Applikation wird für die Speicherung und Übertragung einer schwachen Signatur ein 32-bit großes, vorzeichenloses Integer verwendet. Da die Kollisionsresistent dieser Signatur bei dieser Größe natürlich zu gering ist, gibt es noch die starke Signatur.

2.4 Starke Signatur

Als starke Signatur wird, wie dies auch bei *rsync* der Fall ist, *MD5* verwendet[5]. Diese Hashingfunktion sollte eine genügend niedrige Kollisionswahrscheinlichkeit haben. Dass MD5 als kryptographische Hash-Funktion nicht sicher ist, muss für uns von keinem Belangen sein, da MD5 in keinem sicherheitsrelevanten Kontext verwendet wird.

Für die Einbindung von MD5 in das Programm wird *OpenSSl*[3] verwendet, da dieses eine leicht zu benutzende Implementierung von MD5 anbietet. Weiters ist noch anzumerken, dass der berechnete Hashwert von MD5 zwar nur 16 Byte

groß ist, man beim Speicherverbrauch und Datentransfer allerdings mit 32 Byte Größe rechnen muss, da der Wert für bessere Überprüfbarkeit als 32-stellige Hexadezimalzahl in einem String gespeichert wird.

2.5 Verbesserungsvorschläge

Dieser von mir beschriebene Algorithmus hat einige Verbesserungsmöglichkeiten. So könnte man statt der beschriebenen schwachen Signaturfunktion eine andere mit höherer Kollisionsresistenz verwenden. Tridgell selber schlägt in seiner Arbeit einige andere rollende Signaturen, die sich als schwache Signaturen besser eignen, vor[7]. Als starke Signatur könnte man eventuell auch eine bessere Hash-Funktion, wie etwa *SHA1*, wählen, da bei den heutigen Prozessoren der Unterschied in Rechenkomplexität für den Benutzer vermutlich keinen großen Unterschied machen würde. Mit einer geringeren Kollisionswahrscheinlichkeit könnte man die Blöcke vergrößern – ich schätze heutzutage sind die meisten Netzwerke in der Lage, deutlich größere Datenmengen als 6 KB ohne Probleme zu übertragen –, die Anzahl an zu übertragenden Signaturen verringern und so den Overhead reduzieren.

Der größte Kritikpunkt dieses Synchronisationsalgorithmus ist aber vermutlich die Tatsache, dass der Algorithmus auf der Annahme beruht, dass die lokalen Uhren der beteiligten Geräte synchron sind, damit die Zeitstempel miteinander vergleichen werden können. Um diese Abhängigkeit zumindest teilweise aufzubrechen könnte die Reihenfolge der Ereignisse mit der Lamport-Uhr[2] aufgezeichnet werden.

Die meisten dieser Vorschläge würden aber natürlich die Implementierung des Algorithmus erschweren, weshalb entsprechende Abwägungen bezüglich Vor- und Nachteilen nötig wäre.

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 Synchronisationsablauf

Der Verlauf eines Synchronisationsprozesses zwischen einem Synchronisations-Client und einem Synchronisations-Server 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 entsprechend 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. Alle restlich Dateien, die, die mit ihrem Namen bzw. Pfad sowohl beim Client als auch beim Server vorhanden sind, werden auf Übereinstimmung bezüglich Zeitstempel und Gesamtsignatur überprüft. Für die Dateipaare, bei denen diese Werte 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 die erhaltenen Korrekturdaten zur späteren Dateikonstruktion.

6. Der Server überprüft die starken Signaturen des 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. Für die Stellen, für die es keine Korrekturdaten gibt, wird der Inhalt aus der alten, lokalen Datei verwendet, da diese an dieser Stelle gleich mit der neueren 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 überschrieben.

Ein bedeutender Unterschied zum in Abschnitt 2.1 beschriebenen Algorithmus von *rsync* ist, dass fast alle Entscheidungen vom Server gefällt werden.

Hingegen bei *rsync* werden die Entscheidungen und so auch die Vergleiche der Signaturen auf die beiden beteiligten Netzwerkprozesse aufgeteilt. Diese Ausführung des Synchronisationsverlaufs hat weniger mit Restriktionen in der Verwirklichung als mit einer strukturellen Entscheidung, dass der Server die Deutungshoheit haben sollte, zu tun. Wobei angemerkt werden muss, dass dies zu einer größeren Anzahl an Nachrichten zwischen Client und Server führt.

3.2 Netzwerkkommunikation

Die Netzwerkkommunikation ist mithilfe von *Protocol Buffers*[4] 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 und theoretisch jeder beliebige Wert für ein Byte in seiner serialisierten Nachricht vorkommen kann, kann weder `\n` noch ein anderes Zeichen als Trennzeichen benutzt werden. Daher werden die serialisierten Nachrichten noch zusätzlich in Base 64 kodiert und, da die Menge der 64 Zeichen von Base 64 das Newline-Zeichen nicht enthält, kann dieses ohne Probleme als Trennzeichen benutzt werden. Da die Nachrichten an einem Ende kodiert werden müssen sie dementsprechend auch am anderen Ende wieder dekodiert werden, noch bevor sie deserialisiert werden können. Die Base 64-Kodierung stellt kein besondere Leistungshinderung dar, vergrößert die zu transferierenden Daten aber doch um immerhin ein Drittel. Allerdings ist Base 64 nun Mahl wichtig für die funktionsfähige Interoperation der binären Serialisierung und der textbasierten Netzwerkübertragung.

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, und kann aus mehreren Threads bestehen.

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

3.3.1 Pipe Interfaces

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, welche sich in ihrem Besitz befindet, mit. Dieses kann der File Operator benutzen, um ihnen Rückmeldungen zu schicken.

Um die Unterscheidung von sendenden und empfangenden Enden von Pipes zu ermöglichen, werden Interfaces (rein abstrakte Klassen) und Klassenvererbung benutzt. 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 im Ganzen generisch für einen beliebigen Typ T sind. Soll ein Prozess mit einer Pipe jetzt nur senden bzw. nur empfangen können, so bekommt dieser nur die Schnittstelle `SendingPipe` bzw. `ReceivingPipe` zur Verfügung gestellt.

Nun folgt eine etwas genauere Betrachtung der Implementierung von `Pipe`, da diese schließlich auch Thread-sicher sein muss.

3.3.2 Senden – Pipe::send

```
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)

Hier sehen wir als erste, 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()`, wie in Abschnitt 3.3.4 gezeigt wird, zum Schließen

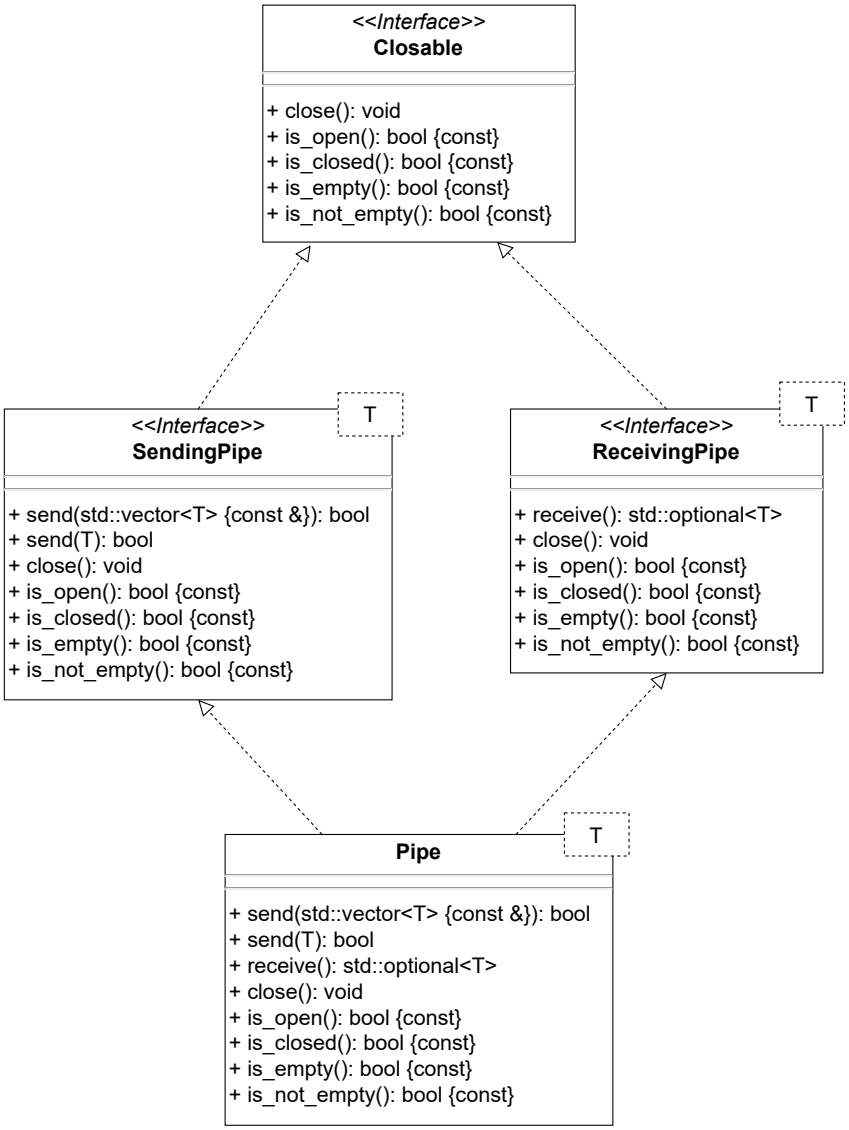

Abbildung 1: UML-Klassendiagramm von Pipe

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 in Abschnitt 3.3.3. 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 ein Thread, welcher auf etwas Empfangbares wartet, 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 mehreren Nachrichten. Diese funktioniert sehr ähnlich. Sie legt alle Nachrichten gleichzeitig in der erhaltenen Reihenfolge in die Queue, sobald dies möglich ist.

3.3.3 Empfangen – Pipe::receive

```
1  std::unique_lock pipe_lck{pipe_mtx};
2  if (is_open()) {
3      // warten, dass es etwas zum Empfangen gibt
4      // oder die Pipe geschlossen wurde
5      receiving_finishable.wait(
6          pipe_lck,
7          [this](){
8              return is_not_empty() || is_closed();
9          }
10     );
11
12     if (is_open()) {
13         T msg{std::move(msgs.front())};
14         msgs.pop();
15
16         return msg;
17     }
18     else {
19         return std::nullopt;
20     }
21 }
22 else {
23     return std::nullopt;
24 }
```

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.

3.3.4 Schließen – Pipe::close

```
1 std::lock_guard pipe_lck{pipe_mtx};
2 open = false;
3 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 Thread gerade zufällig überprüft, ob er mit dem Warten aufhören kann, weil das Mutex ist 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 somit im wartenden Status verbleiben. Aus diesem Grund muss close() das Mutex auch sperren.

Damit haben wir uns nun einen kleinen Überblick über ein paar wichtige und interessante Teile des Programms verschafft und es geht weiter zur Erklärung, wie man das Programm benutzt.

4 Bedienung

Es gibt viele verschiedene Möglichkeiten die Applikation zu bedienen. Parameter können mittels CLI, Umgebungsvariablen oder einer Konfigurationsdatei im Format *JSON* bestimmt werden. Weiters kann man, sobald das Programm gestartet ist, über eine Kommandozeile noch zusätzlich mit dem System interagieren.

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 einen Systemprozess zu starten, gibt man beide Kommandozeilenargumente an.

Beim Synchronisationsverzeichnis sollte sicher gegangen werden, dass das Programm die entsprechenden Rechte hat, um alle Dateien lesen und beschreiben zu können. 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 hier erstellt und beschrieben werden. Außerdem werden Dateien im Unterverzeichnis *.sync* nicht synchronisiert.

4.1 Konfiguration

Das Programm `sync` kann wie bereits erwähnt auf verschiedene Weisen konfiguriert werden. Die konkrete Hierarchie, in der die Konfigurationswerte angewendet werden, sieht wie folgt aus:

1. Es gibt *Standardwerte* für die meisten Parameter. Diese werden überschrieben durch gesetzte
2. *Umgebungsvariablen*, welche wiederum überschrieben werden durch die Werte in der
3. *JSON Konfigurationsdatei*, wenn diese bereitgestellt wurde, und das letzte Wort hat die
4. *Kommandozeilenschnittstelle* (englisch: command line interface; kurz: CLI)

Alle konkreten Einstellungsmöglichkeiten inklusive Beschreibungen und auch Beispielen lassen sich in der *README* des Projekts nachlesen.

4.2 Kommandozeile

Sobald man die Applikation gestartet hat, bekommt man Zugriff auf eine Kommandozeile, mit welcher man zusätzlich mit dem Synchronisationssystem interagieren kann. Diese Kommandozeile bietet alle Annehmlichkeiten, welche man sich von einer grundlegenden Kommandozeile erwartet. So kann man zum Beispiel mit den Pfeiltasten nach oben und unten die bereits benutzten Befehle durchgehen. Um diese Kommandozeile zu verlassen und das Programm zu beenden, kann man entweder einen der äquivalenten Befehle `q`, `quit` oder `exit` oder das Tastaturkürzel `Strg+D` benutzen.

Die folgende Tabelle zeigt alle Befehle, die in der Kommandozeile aufgerufen werden können, mit Beschreibungen:

Befehl	Beschreibung
`h`, `help`	Gibt eine ein Hilfe aus
`ls`, `list`	Listet alle Dateien, die synchronisiert werden, auf
`ll`, `list long`	Listet alle Dateien, die synchronisiert werden, inklusive Signatur, Größe und Zeitpunkt der letzten Änderung auf
`sync`	Startet, wenn möglich, einen Synchronisationdurchlauf mit dem Server und ladet alle die Informationen über alle Dateien neu
`q`, `quit`, `exit`	Beendet und verlässt die Anwendung

4.3 Protokollierung

Die Ausgabe des Protokolls (englisch: Log) auf die Konsole kann mit dem Kommandozeilenargument `-l` aktiviert werden und mit `--log-level` kann man einstellen, ab welchem Level (0 ... Verfolgung bis 5 ... kritisch) die protokollierten Nachrichten angezeigt werden sollen. Weiters kann man das Protokoll auch in eine Datei schreiben lassen. Der Dateiname für diese Datei kann mit dem Kommandozeilenargument `-f` angegeben werden. Existiert die angegeben Datei noch nicht, wird sie vom Programm erstellt. Außerdem wird die angegebene Protokolldatei beim Synchronisieren ignoriert. Es ist empfehlenswert den Kommandozeileparameter `--no-color` zu setzten, wenn man das Protokoll in eine Datei schreibt, da ansonsten die Kontrollzeichen für die farbige Ausgabe in die Datei geschrieben werden, was das Protokoll schwerer lesbar macht. Weitere Einstellungsmöglichkeiten für die Protokollierung und die Protokollausgabe lassen sich in der *README* des Projekts nachlesen.

Literatur

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

[2] Leslie Lamport. „Time, Clocks, and the Ordering of Events in a Distributed System“. In: *Commun. ACM* 21.7 (Juli 1978), S. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

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

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

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

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

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