Rust Projekt Cryptominer Teil 2

Poolanbindung

Einleitung

Aufgrund der steigenden Beliebtheit von Cryptowährungen nimmt auch die Anzahl der „Miner“ immer weiter zu. Da das Bitcoin Netzwerk und auch andere Plattformen, die Schwierigkeit Transaktionen zu validieren (einen Block zu minen), der „Rechenpower“ des gesamten Netzwerks anpassen, ist es heutzutage unmöglich alleine einen Block zu finden. Aufgrund dessen schließen sich „Miner“ in sogenannten „Miningpools“ zusammen. Dabei arbeiten mehrere Leute zusammen an dem „Proof of Work“-Problem. Wenn ein Block an Transaktionen validiert wird, wird der „Miner“ vom Netzwerk für seine Leistung in der Währung des Netzwerks belohnt. Im Falle von Bitcoin erhält der Miner „Bitcoins“ Diese Bitcoins werden beim Zusammenschluss im Pool einfach auf den gesamten Pool verteilt. Dadurch haben „Miner“ konstantere Einnahmen und können auch Geld verdienen ohne einen Block zu finden.

Umsetzung im Projekt

Da der von uns entwickelte „Mining-Algorithm“ in Rust relativ unspektakulär und somit auch langsam ist, ist eine Poolanbindung hierbei dringend von Nöten. Für die Umsetzung haben wir uns hierbei für den MiningPool „Slushpool“ entschieden. Dies ist einer der größten MiningPools und hat durch die vielen User eine hohe Wahrscheinlichkeit einen Block zu finden.

Die Poolanbindung für den Algorithmus erfolgt über einen TCP-Stream. Die Kommunikation zwischen „Miner“ und Pool wird durch ein speziell hierfür entwickeltes Protokoll, namens „Stratum“ unterstützt.

Dieses Protokoll funktioniert nach folgendem Schema:

1. Der User sendet eine „Subscribe-Nachricht“ an den Server und teilt ihm somit mit, dass er an Daten zum Lösen von Blöcken interessiert ist.
2. Der User sendet eine „Authentifizierung-Nachricht“, in welcher er seine Arbeiter vorstellt, die die Aufgaben lösen werden.
3. Der Server bestätigt die Authentifizierung und die „Subscription“
4. Der Server sendet Daten zum Minen und wartet gleichzeitig auf Antworten
5. Wenn der „Miner“ eine Lösung findet kann er diese beim Pool einreichen. Ansonsten bekommt er immer weiter Aufgaben und kann sich aussuchen welche er davon bearbeiten möchte.

Da der „Miner“ sozusagen mit Aufgaben überschüttet muss ein Miner eine sogenannte Scheduling-Strategie implementiern. Dabei muss er intern festlegen, welche Aufgaben als erstes bearbeitet werden. Unser Miner geht hierbei relativ stupide nach dem FIFO-Prinzip vor.

Verbindungsaufbau und Jobverteilung

Beim startend es Programms wird zu allererst eine Verbindung wie oben beschrieben,zum Server aufgebaut. Dazu wird ein TCP-Stream geöffnet und mit einem „BufferedReader“ ausgelesen. Diese Verbindung wird von der „connection.rs“-Datei gehandelt.
Beim erhalten einer Nachricht werden diese Daten mit dem crate „serde_json“

konvertiert und im Anschluss in ein Job Struct umgepackt. Dieser Struct wird dann in einer Warteschlange gespeichert. Bei einem asynchronen Ansatz würden einzelne Threads diese Jobs dann abgreifen und das „Proof-of-Work“ Problem lösen. Aufgrund des Zeitmangels konnten wir jedoch keine Threads implementieren, weshalb der Mining Job nach Erstellung direkt an den Algorithm übergeben wird. In dieser Zeit wird der „Buffered-Reader“ also auch blockiert, was suboptimal ist! Nach dem der Algorithmus die Aufgabe gelöst hat bekommt das Programm die Lösung oder ein None mittels einer „Option“ zurück. Falls eine Lösung vorliegt wird diese an der Server gesendet, falls nicht wird der Job einfach gelöscht und ein neuer Bearbeitet.

Fazit

Abschließend nach dem Projekt lässt sich sagen, dass wir dieses durchaus unterschätzt haben. Die Kombination aus der für uns neuen Programmiersprache Rust, dem speziellen Thema, zu dem es wenig konkrete Implementierungen in allen Sprachen gibt, und dem damit verbunden Rechercheaufwand hat den zeitlichen Aufwand selbst für einfache Funktionen explodieren lassen. Der Rechercheaufwand betrug für die Mining-Komponente circa die dreifache Zeit im Vergleich zu Implementierung, ungeachtet des Debuggens oder Recherchieren für Rust. Alles in allem würden wir ein anderes Thema wählen, nicht des Themas sondern des Rechercheaufwands wegen. Ein Thema wobei mehr Code entsteht und der Fokus mehr auf dem Implementieren liegt wäre für zukünftige Projekte dieser Form sicherlich die bessere Wahl.

Zu den größten Herausforderungen abgesehen vom genannten gehörte wie in allen Projekten die Abstimmung der Aufgabenbereiche untereinander. Da uns diese Schwierigkeit bereits in anderen Projekten begegnet ist haben wir zu Beginn grobe Konstrukte und Datenschnittstellen zwischen den Komponenten sowie die zu verwendenden Datentypen definiert, welche zur Kommunikation und dem einheitlichen Aufbau der Module dienen sollten. Darüber hinaus haben wir fast ausschließlich in VSCode mit der Live-Funktion zusammen am Projekt gearbeitet was den Fortschritt durch bessere Kommunikation beschleunigte. Auch mit Rust selbst hatten wir zu Beginn Probleme vor Allem bezüglich des Ownership-Modells, welches in den komplexen Datenstrukturen und verzweigten Funktionen

nicht immer klar ersichtlich war. Gegen Ende des Projekts fiel uns die Anwendung jedoch immer leichter und die Strenge des Compilers wurde mehr als Vorteil wahrgenommen.