Einleitung

Da die vertikale Skalierung von Rechenressourcen aufwändiger wird, wird das nutzen von verteilten Rechenressourcen zunehmend wichtiger. Anstatt ein Rechenproblem einer einzelnen rechenstarken Maschine zu geben, kann die Rechenkraft vieler Maschinen (Nodes) genutzt werden. Dafür ist es wichtig die Aufgabe in Teilaufgaben zu unterteilen und an die Nodes zu verteilen. Dies wird oft über Messaging Protokolle gelößt.

Im Rahmen des Forschungsprojekts "Grischa" an der HTW Berlin wird an Verteilten Systemen geforscht. Grischa ist ein Schachprogramm, welches, anders wie Moderne Schachcomputer, nicht mit Heuristiken oder KI rechnet, sondern versucht möglichst viele Züge im Vorraus zu berechnen ([TODO]: Details aus MA nehmen!) und zu evaluieren. Für die Kommunikation zwischen den verteilten Rechenressourcen wird derzeit die Messaging Bibliothek ZeroMQ benutzt. Ziel dieser Arbeit ist die Analyse von Grischas Messaging-System sowie die Evaluation von ZeroMQ mittels Gegenüberstellung zu einer alternative namens nanomsg in den Punkten Geschwindigkeit und Bedienbarkeit.

Grischa

Grischa wurde historisch mit verschiedenen Kommunikationsstrategien betrieben, z.B. mit Redis. Die für diese Arbeit relevante Implementierung, mit ZeroMQ, besteht aus 3 Teilanwendungen:

1. "GClient": Zuständig für die Koordinierung der Aufgaben sowie Schnittstelle zum abgreifen der errechneten Züge z.B. in der implementierten variante ([TODO]: Welches programm war das nochmal?)
2. "GNode": Stellt die Worker-Node dar. Nimmt aufgaben vom GClient entgegen und bewertet Spielzüge. Je mehr GNode Instanzen erstellt werden, desto schneller können Spielzüge berechnet werden.
3. "GBroker": Stellt den Message Broker dar, der die Kommunikation zwischen den Anwendungen GClient und GNode handhabt. Zwar kann ZeroMQ auch ohne Message Broker arbeiten, er erleichtert jedoch Anmeldung und Ausfall von Nodes. Gleichzeitig können GNodes über den GBroker ihre Ergebnisse zurück geben.

Für diese Arbeit ist die Message-Basierte Kommunikation von vorrangiger Bedeutung. Dafür wird ZMQ an zwei Stellen eingesetzt.

1. In der Kommunikation zwischen den GNodes und dem GClient. Dies wird umgesetzt mit Sternförmiger Topologie mit dem Router des GBroker als zentrales Element sowie dem XSubXPub Modell. Vorteil von PubSub sind die Topic-gesteuerten Nachrichten via URI Adressen. Mit einem Broker in der Mitte wird daraus XSUBXPUB. Wenn man nun für jede Datenflussrichtung eine Verbindung erstellt, hat man eine bidirektionale Verbindung welche wichtig ist damit jedes Modul mit jedem anderen Modul komminizieren kann. [Ros 43]
2. In der Kommunikation zwischen den internen Modulen der GNode Anwendung.

- Wie funktioniert Grischa allgemein?
    - typen
- Was macht es besonders? / anders?

Messaging

- Messaging:
    - Protokolle
        - Scalability Protocolls
    - Libraries
        - ZeroMQ/nanomsg
            - unterschiede, gemeinsamkeiten
        - notable mentions:
    - Strukturen
        - Pub/Sub
        - XPub/XSub
        - Req/Rep / XReq/XRep

Analyse

- Grischas Kummunikationsmodell
    - Helgers Arbeit analysieren / wiederholen
    - Grafiken um Datenfluss zu visualisieren

Planung

Relevant für die Grischa Anwendung ist weniger die Echtzeit-Verarbeitung und mehr die Skalierbarkeit auf viele GNodes mit einem weiterhin Stabilen Datendurchsatz. Da in Grischa sowohl das TCP als auch das IPC Protokoll genutzt wird, werden beide Protokolle untersucht. Desweiteren wird untersucht wie sich der Durchsatz im Verhältniss zur Nachrichtengröße verhält. Diese Parameter sollen sowohl auf dem bisherigen System ZeroMQ als auch auf der alternative nanomsg umgesetzt werden.

Zusammengefass sind die untersuchten Parameter sind :

- Message-Bibliothek (ZeroMQ und nanomsg)
- Nachrichtengröße (msg_size)
- Anzahl der Nodes (Clients)
- Protokoll (TCP und IPC)

Um dem Anwendungsfall von Grischa nachzustellen, wird ebenfalls mit Master- (Client), Router- und Worker-Anwendungen gearbeitet. Da das PubSub-Modell die Komplexität durch das Topic-basierte abbonieren von Nachrichten erhöht, wird auf ReqRep zurückgegriffen (bzw. "XReqXRep").

Es soll sichergestellt sein dass der einzige Unterschied zwischen der ZeroMQ und der nanomsg Implementierung lediglich die Bibliotheksrelevante Schnitstelle ist. Die Business-Logik muss genau gleich sein, um unterschiede hier auszuschließen. Die Anwendungen sollte daher auch separat kompelliert werden und nicht intern entscheiden welche Bibliothek ausgeführt wird. Außerdem sollten Unterschiede durch Compileroptimierungen ausgeschlossen werden indem die gleiche Programmiersprache mit den gleichen Compileroptionen verwendet werden.

- Motivation/Ziel: ?

- Warum ReqRep?
    - Verlässlichkeit
    - Einfachheit

Durchführung

Als Programmiersprache für die Implementierungen wurden vorrangig die nativen implementierungen von ZeroMQ und nanomsg in Betracht gezogen, damit einem Performanceverlust durch Übersetzungsschichten vorgebeugt wird. ZeroMQ ist nativ in C++ (libzmq), nanomsg in C geschrieben. Da die Arbeit an ZeroMQ bereits 2007 angefangen wurde sowie deutlich populärer ist und Version 1.0 von nanomsg erst Mitte 2016 erschienen ist, gibt es mehr Langugage-Bindings ([TODO]: Deutsche wörter finden) in C. Da man daher davon ausgehen kann dass ZeroMQ ausgereifter ist, wurde sich auf C als gemeinsame Programmiersprache festgelegt. Bei nanomsg kann so die native bibliothek benutzt werden, bei ZeroMQ wird czmq als High-Level Language-Binding ([TODO]: same: schöne Deutsche wörter finden) Bibliothek genutzt.

Router/Broker

Der Router beschränkt sich, sowohl bei ZeroMQ als auch bei nanomsg, auf sehr wenig Code. Daher wurde hier jeweils ein eigenes Programm für beide Bibliotheken implementiert.

Client (Master)

Zuerst generiert der Worker eine zufällige Zeichenkette zum übertragen mit der Länge von 2^i, wobei i bis zum angegebenen $max_msg_size_power$ läuft. In die gleiche Nachricht werden die Metainformationen aus Tabelle x ([TODO]: Nummer updaten) als "Header" eingefügt ohne die größe zu verändern. Danach wird die Nachricht mit zmq bzw. nanomsg abgesendet. Dafür wird in beiden fällen eine Art Kontext/Socket erstellt und sich mit dem Broker verbunden. Dann wird die Nachricht abgesendet, die Antwort erwartet und die antwort gelöscht. Dieser Senden/Empfangen Prozess wird $repetitions$-mal wiederholt. Danach werden die Bibliothektsspezifischen Daten gelöscht. Danach wird i, und danach die damit erhöhte msg_size vielfach abgeschickt.

Worker

Der Worker ist für das Benchmarking besonders wichtig. Zuerst initialisiert er die Verbindung zum Broker via nanomsg bzw. ZeroMQ. Dann stoppt er die Zeit zum Empfangen, Verarbeiten und Antworten mit der Anzahl von $repetitions * clients$ (aus dem Header der Nachricht). Daher werden mit zunehmenden Clients, zunehmend mehr. ([TODO]: Code, warum nicht beim Senden der nachricht repetitions/clients teilen? Dann ist die Problemgröße gleichgroß).

Header der gesendeten Nachrichten.

Inhalt	**Beschreibung**
tag	Enthält Flags, die zur Steuerung des Workers notwendig sind
client_id	Enthält die numerische ID des absenders, maßgeblich für debugging Zwecke

Inhalt	Beschreibung
msg size	Übermittelt die msg Größe, wenngleiche die tatsächlich Länge durch `strlen()` ermittelt wird
repetitions	Enthält die Anzahl der Wiederholungen die in dieser msg_size vorliegen. I.d.r 5000
client_count	Beschreibt wie viele clients anfangs gestartet wurden

Um

- ...
- Was wurde alles getestet?
    - Protokolle: IPC/TCP
    - message size: 4B*???KB
    - zeromq (czmq) & nanomsg
- Erwartungshaltung

Ergebnisse

- Überraschende Ergebnisse
- Objektiv darstellen was man sieht

Diskussion

- Was bedeuten die Ergebnisse?
    - Warum ist es hier langsamer / schneller
    - Aspekte:
        - msg size (was ist für Grischa relevant)
        - client count: ab wann wird es langsam? Warum? Was könnte man ggf. dagegen tun?
- Was weiß man nicht? Wo muss noch nachgearbeitet werden?
- Was heißen die Ergebnisse im Kontext von der Grischa Arbeit?
- Habe czmq anstelle von libzmq genutzt -> das wäre noch besser und hätte auch ein vergleichbareres interface

Zusammenfassung

Ausblick

META

TODO

- auf einheitliches wording (z.B. zeromq) achten