

Studienarbeit II

Entwicklung eines interaktiven Verbindungsmanagementsystems

Erweiterung der Studienarbeit I mit Handshake-Protokollen für dezentrale
Go-Anwendungen

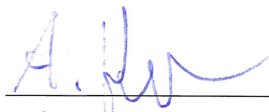
vorgelegt am 17. Februar 2026

Name:	Alexander Betke, Theo Leuthardt
Matrikelnummer:	77203378972, 77205844868
Ausbildungsbetrieb:	Polizei Berlin, Bundesdruckerei GmbH
Studienjahrgang:	2023
Fachbereich:	Duales Studium Wirtschaft · Technik
Studiengang:	Informatik
Betreuer/in Hochschule:	(Prof. Dr.) Arthur Zimmermann
Anzahl der Wörter:	6000

Vom Ausbildungsleiter zur Kenntnis genommen:



AL Cynthia Reuter



Alexander Betke

AL Sven Krausch

Theo Leuthardt

Abstract

In heutigen verteilten Systemen stellt die benutzerorientierte Herstellung einer Netzwerkverbindung eine zunehmende Komplexität dar. Automatisierungen hinsichtlich des Verbindungsaufbaus reduzieren diese Komplexität erheblich, jedoch wird damit auch ein Teil der Kontrolle über die Netzwerktopologie abgegeben.

Die in Studienarbeit I entwickelte Desktop-Anwendung mit der Programmiersprache Go zur Visualisierung des Leser-/Schreiber-Problems wird im Rahmen dieser Arbeit um ein Interaktives Verbindungsmanagementsystem erweitert. In der vorherigen Version der Desktop-Applikation werden andere Instanzen der Anwendung im lokalen Netzwerk gesucht und es wird eine Verbindung mit der erst möglichen Instanz automatisch hergestellt. Mit der Erweiterung im Zuge dieses Projekts wird dem Benutzer ermöglicht selbst eine beliebige Instanz im lokalen Netzwerk auszuwählen und eine Verbindung aufzubauen.

Zur Umsetzung dieser Erweiterung werden eigen entwickelte TCP-Handshake-Protokolle entwickelt auf Basis des TCP-Handshake-Protokolls. Das Backend wird mit jenen Protokollen zur Discovery anderer Peers ergänzt inklusive neuen Implementierungen für den Verbindungsaufbau. Zur Anzeige aller aktuell verbindungsreifen Peers im lokalen Netzwerk wird dem Frontend eine Übersicht hinzugefügt, mit der der Benutzer einen Peer zum Verbindungsaufbau auswählen kann. Für die saubere Trennung einer bestehenden Verbindung werden im Backend Graceful Disconnections implementiert.

Schreibverteilung

Autor	Kapitel
Theo Leuthardt	Abstract Einleitung Problemstellung Anforderungsanalyse Entwurf - Frontend Algorithmen - Frontend Fazit
Alexander Betke	Theoretische Grundlagen Entwurf - Frontend Algorithmen - Backend Diskussion

Inhaltsverzeichnis

Abstract	I
Inhaltsverzeichnis	III
Abbildungsverzeichnis	IV
Codeverzeichnis	V
Akronyme	VI
1 Einleitung	1
2 Problemstellung	2
2.1 Problemrelevanz	2
2.2 Fehlende Funktionalitäten	3
3 Anforderungsanalyse	4
3.1 Funktionale Anforderungen	4
3.1.1 Peer-Discovery	4
3.1.2 Verbindungsverwaltung	4
3.1.3 Verbindungstrennung	5
3.2 Nichtfunktionale Anforderungen	5
3.2.1 Performance	6
3.2.2 Benutzerfreundlichkeit	6
3.2.3 Zuverlässigkeit	6
3.2.4 Skalierbarkeit	6
4 Theoretische Grundlagen	7
4.1 Peer-Discovery in lokalen Netzwerken	7
4.2 Anwendungsschicht-Handshake als Endpunktvalidierung	7
4.3 Nebenläufigkeit im Kontext der Netzwerkkommunikation	8
5 Entwurf	10
5.1 Strukturelle Veränderungen im Backend	10
5.2 Strukturelle Veränderungen im Frontend	10
6 Algorithmen	12
6.1 Implementierungsübersicht	12
6.2 Portreservierungsalgorithmus	12
6.3 Discovery-Protokoll	12

6.4	Verbindungsaufbau	14
6.5	Backend - Frontend Kommunikation	15
6.6	Button-Grid Layout für Peer-Auswahl	16
6.7	Dynamische Peer-Button-Verwaltung	17
6.8	Zustandsabhängige UI-Sichtbarkeit	18
6.9	Sicherheits- und Netzwerkeffekte	19
7	Diskussion	20
7.0.1	Alte Version: Ohne explizite Peer-Auswahl	20
7.0.2	Neue Version: Mit Peer-Auswahlliste	20
7.1	Auswirkungen der neuen Peer-Auswahl	21
8	Fazit	22
	Literaturverzeichnis	23
	KI-Verzeichnis	25
	Ehrenwörtliche Erklärung	26

Abbildungsverzeichnis

1	Schematischer Ablauf der Peer-Prüfung in <code>utils.IsPeerDiscoverable</code>	13
2	Alte Version: Fehlende Peer-Discovery-Visualisierung	20
3	Neue Version: Explizite Peer-Discovery-Liste mit Auswahlmöglichkeit	21

Codeverzeichnis

6.1	Portreservierung (Code wie im Repository)	12
6.2	Peer-Discovery (vereinfacht, sequentiell)	14
6.3	Vereinfachte Behandlung der <code>CONNECT_REQ</code> -Nachricht (Ausschnitt aus <code>handleConnection</code>)	14
6.4	Backend: Periodisches Senden des <code>QueueState</code> (vereinfacht)	15
6.5	Frontend: Empfang und Verarbeiten der Zustandsnachrichten (vereinfacht)	15
6.6	Frontend: Senden eines Verbindungswunsches (vereinfacht)	16
6.7	Backend: Empfang von Steuer-Signalen (vereinfacht)	16
6.8	Initialisierung des Button-Grid Containers	16
6.9	Einbettung in Scroll-Container	17
6.10	Dynamische Button-Verwaltung (Ausschnitt aus <code>updateUI</code>)	17
6.11	Zustandsabhängige UI-Aktualisierung (Ausschnitt)	18

Akronyme

KI Künstliche Intelligenz

Glossar

Nebenläufigkeit Zwei Prozesse A und B heißen nebenläufig (concurrent), wenn sie voneinander unabhängig bearbeitet werden können. Für die Realisierung der Bearbeitung von A und B gibt es also die folgenden drei Fälle. 1. Zuerst A, dann B. 2. Zuerst B, dann A. 3. A und B gleichzeitig. [Wagenknecht, 2004, S. 224]

1 Einleitung

Verteilte Systeme ermöglichen die effiziente Bearbeitung komplexer Aufgaben durch eine potentiell effiziente Aufteilung auf mehrere bearbeitende Instanzen. Bei zunehmender Komplexität solcher Aufgaben werden Protokolle benötigt, um einen sicheren Datenaustausch zwischen den einzelnen Bearbeitern zu gewährleisten. Um die Aufteilung von Aufgaben auf Teilnehmer verteilter Systeme zu erforschen und das Leser-/Schreiber-Problem zu visualisieren wird in der vergangenen Studienarbeit I eine Applikation in Go entwickelt, mit der zwei Instanzen dieser Anwendung in einem lokalen Netzwerk eine Verbindung aufbauen können per TCP-Protokoll und anschließend Pakete gegenseitig aneinander versendet werden.

Mit dieser Arbeit wird die bestehende Implementierung erweitert um ein Verbindungsmanagementsystem. Dieses soll dem Nutzer die Möglichkeit geben selbst die Instanz auszuwählen, mit der sich die Anwendung verbinden soll. Für die Erweiterung wird im Frontend eine Übersicht aktuell vorhandener und verbindungsbereiter Instanzen entwickelt zum interaktiven Aussuchen des Verbindungspartners. Dafür wird auf Basis des TCP-Handshake-Protokolls ein eigenes Handshake-Protokoll entworfen und für saubere Verbindungstrennungen Graceful-Disconnections implementiert. Da dies eine Erweiterung des vorhandenen Programmcodes ist, wird die Programmiersprache GoLang für das Frontend und Backend beibehalten.

Im folgenden Kapitel werden mit der 2 die noch vorhandenen Probleme aus Studienarbeit 1, die Problemrelevanz und noch fehlende Funktionalitäten beschrieben. Die daran anschließende Anforderungsanalyse konkretisiert die funktionalen und nicht funktionalen Anforderungen, die sich daraus ergeben. Für das Verständnis der späteren Kapitel wie dem 5 zu architektonischen Anpassungen oder 6 mit den konkreten Implementierungen wird im Kapitel 4 jegliche konzeptionelle Grundlage erläutert. Mit der 7 werden die Ergebnisse des Projekts evaluiert und von einem kritischen Standpunkt betrachtet. Abschließend wird in Kapitel 8 ein Fazit gezogen und zukünftige mögliche Änderungen bzw. Verbesserungen angeführt.

2 Problemstellung

Das mit der Studienarbeit I implementierte Softwaresystem in Form einer Desktop-Anwendung ermöglicht es TCP-Verbindungen zwischen zwei Instanzen innerhalb eines lokalen Netzwerks vollautomatisch herzustellen. Erkennt das Backend einer Instanz eine andere wird direkt eine Verbindungsanfrage gesendet. Die Automatisierung dieses Prozesses führt zu den drei folgenden Beschränkungen:

1. Es kann nicht bestimmt werden, mit welchem Peer die Verbindung aufgebaut wird.
2. Es existiert keine Funktion zum manuellen Trennen einer bestehenden Verbindung.
3. Der Anwender erhält keinerlei Information darüber, welche weiteren Peers im Netzwerk verfügbar sind oder mit welchem konkreten Peer aktuell kommuniziert wird.

Falls mehr als zwei Instanzen in einem lokalen Netzwerk ausgeführt werden, ist es mehr Zufall als gewolltes Szenario der Benutzer welche Instanz mit welcher anderen eine Verbindung eingeht. In der Regel sendet eine Instanz bei beispielsweise drei Instanzen in einem lokalen Netzwerk an eine zweite eine Verbindungsanfrage und verbindet sich mit dieser unabhängig davon, ob dies vom Nutzer so gewollt ist. Zum Wechseln des Peers mit dem die Verbindung eingegangen werden soll muss die eigene Instanz vollständig beendet werden. Nur so ist es bisher möglich die Verbindung zu trennen.

2.1 Problemrelevanz

Die manuelle Verbindungssteuerung wird in modernen verteilten Systemen wie Microservices oder verteilten Netzwerken über mehrere Rechenzentren als essentiell angesehen. In manchen Szenarien solcher Produktivsysteme wird die bewusste Auswahl des Verbindungspartners erfordert für das Trouble-Shooting fehlerhafter Teilkomponenten oder die Kommunikation mit eindeutigen Services. Im Falle der Visualisierung des Leser-/Schreiber-Problems zum Erlernen für Studierende ist es notwendig verschiedene Szenarien durcharbeiten zu können zum besseren Verständnis des Problems, wodurch das spezifische Auswählen des Kommunikationspartners benötigt wird. Wird ein Anwendungsbeispiel wie die Verbindung mit Testinstanzen für eine Entwicklungs- bzw. Testumgebung betrachtet, wird die zu aufwendende Zeit für das Testen durch das wiederholte Neustarten der Anwendung erhöht, wodurch ein ineffizienterer Testablauf verursacht werden kann.

2.2 Fehlende Funktionalitäten

Nach einer Analyse der bestehenden Implementierung der Desktop-Anwendung werden sowohl funktionale als architektonische Defizite in der Frontend- und Backend-Schicht ersichtlich. Architektonisch wird eine Diskrepanz erkannt zwischen dem Informationsgehalt im Backend über die gefundenen Peers im Backend durch automatische Peer-Discovery und die fehlenden Informationen darüber im Frontend für eine Übersicht des möglichen Verbindungspartner.

In der Backend-Schicht der Implementierung wird automatisch nach anderen Peers im lokalen Netzwerk gesucht mit den Ports 50500 bis 50600. Nach einer erfolgreichen Verbindung mit einem anderen Peer, wird aber der Scan-Algorithmus beendet und erst wenn die Anwendung neu gestartet wird, wird erst wieder nach neuen Peers gesucht. Sollte also ein Peer nicht auch schon verbindungsbereit im selben Netzwerk vorhanden sein, kann eine Verbindung erst nach einem Anwendungsneustart hergestellt werden. Außerdem werden durch das Backend zwar bei der Verbindungstrennung die grundlegenden Handshakes unterstützt durch `CONNECT_REQ` und `CONNECT_OK` Nachrichten, jedoch wird keine Implementierung bereitgestellt für Graceful-Disconnects zur vollständig koordinierten Verbindungstrennung.

Zusätzlich wird dem Nutzer im Frontend nicht ersichtlich gemacht, welche anderen Peers gerade zur aktuellen Laufzeit verbindungsbereit vorhanden sind. Ebenfalls wird keine Benutzerinteraktionsmöglichkeit angezeigt wie eine Schaltfläche zum manuellen Trennen einer aktuell vorhandenen Verbindung. Durch diese beiden fehlenden Funktionalitäten wird eine Informationslücke für einen kontrollierten Anwendungsablauf dargestellt.

3 Anforderungsanalyse

Mit der Erweiterung der vorhandenen Implementierung aus Studienarbeit I werden Definitionen von Anforderungen gefordert. Damit wird eine Grundlage gebildet, auf der der Entwurf und die Implementierung des Verbindungsmanagementsystems umgesetzt werden kann. Es wird als notwendig betrachtet, dass alle Grundfunktionalitäten aus der Studienarbeit I beibehalten werden. Dazu wird auf funktionaler Seite das automatische Erkennen von Peers im lokalen Netzwerk gezählt, sowie die Nutzung des Portbereichs 50500 bis 50600, das gegenseitige Versenden von TCP-Paketen, die Anzeige und das Monitoring des aktuellen Verbindungsstatus, die Visualisierung des Leser-/Schreiber-Problems und die Channel-basierte Kommunikation zwischen Frontend und Backend. Bei den nicht funktionalen Anforderungen werden die Plattformunabhängigkeit der Anwendung, UI-Skalierung und der Dark-Mode als vorhanden gesehen.

In den folgenden Teilabschnitten werden die Anforderungen in funktionale und nicht funktionale unterteilt und darin jeweils in kategorisierte Untergruppen. Jede definierte Anforderung wird mit einer eindeutigen Identifikationsnummer notiert für die spätere Referenzierung in Kapitel 5 und 6.

3.1 Funktionale Anforderungen

In folgender Auflistung werden die neuen funktionalen Anforderungen beschrieben für das erwartete Verhalten der Desktop-Anwendung.

3.1.1 Peer-Discovery

FA-01: Peer-Liste-Verwaltung

Zusätzlich zur automatischen Erkennung von Peers im lokalen Netzwerk soll das Backend eine Liste an erkannten Peers führen und diese regelmäßig bei jeder neuen Suche aktualisieren.

FA-02: Discovery-Handshake

Damit auch nur valide Peers erkannt werden, soll vor dem eigentlichen Verbindungs-Handshake eine Erkennungs-Handshake implementiert werden in Form von: `DISCOVER_SYN` / `DISCOVER_ACK`. Dies soll zur Prüfung dienen, ob der gefundene Peer verfügbar für eine anschließende Verbindung ist.

3.1.2 Verbindungsverwaltung

FA-03: Manuelle Peer-Auswahl

Bei der Nutzung der Anwendung soll der Benutzer über eine grafische Benutzeroberfläche ei-

ne Übersicht über alle aktuell erkannten Peers erhalten, mit denen potentiell eine Verbindung aufgebaut werden kann.

FA-04: Peer-Auswahl-Ausblendung im verbundenen Zustand

Wird eine Verbindung hergestellt mit einem anderen Peer, so soll die Peer-Auswahl-Übersicht aus Anforderung 3.1.2 ausgeblendet werden.

FA-05: Verbindungs-Handshake-Protokoll

Für das Verbinden mit einem Peer soll ein Handshake-Protokoll implementiert werden mit `CONNECT_REQ:<ip>:<port>` und `CONNECT_OK` Nachrichten.

FA-06: Bidirektionale Informationsaustausch über die Verbindung

Bei einer Verbindungsanfrage über die `CONNECT_REQ` Nachricht soll die IP-Adresse und der Port des Verbindungsinitiators mit versendet werden, damit auch der Empfänger die Information erhält, wer mit ihm eine Verbindung aufbauen möchte.

FA-07: Verbindungsexklusivität

Wird zwischen zwei Peers eine Verbindung erfolgreich aufgebaut und erhalten, sollen Verbindungsanfrage-Nachrichten anderer Peers währenddessen ignoriert werden.

3.1.3 Verbindungstrennung

FA-08: Manueller Disconnect

Dem Benutzer der Anwendung soll es über eine Schaltfläche ermöglicht werden, eine bestehende Verbindung zu einem anderen Peer trennen zu können, ohne die Anwendung schließen zu müssen.

FA-09: Graceful-Disconnects

Wird eine bestehende Verbindung zu einem anderen Peer manuell unterbrochen durch den Benutzer, soll eine `DISCONNECT` Nachricht erst an den Peer gesendet werden für die Verbindungsauflösung.

FA-10: Zustandssynchronisation bei Disconnect

Nachdem durch den Benutzer die Verbindung zu einem anderen Peer getrennt wird, sollen beide Instanzen ihre Zustände wie der Verbindungsstatus, die Adresse des verbundenen Peers und die Queue-Werte zurücksetzen auf ihre Initialwerte.

3.2 Nichtfunktionale Anforderungen

Im Folgenden werden die nicht funktionalen Anforderungen definiert zur Hervorhebung qualitativer Eigenschaften der Anwendung.

3.2.1 Performance

NFA-01: Discovery-Geschwindigkeit

Ein vollständiger Discovery-Scan (localhost + Subnetz) soll innerhalb von maximal 5-7 Sekunden abgeschlossen sein.

NFA-02: UI-Aktualisierungsrate

Die Benutzeroberfläche muss im Intervall von 100ms mit aktuellen Zustandsinformationen aus dem Backend aktualisiert werden.

NFA-03: Handshake-Timeout

Discovery-Handshakes sollen einen Timeout von 2 Sekunden haben und die Verbindungs-Handshakes einen Timeout von 1 Sekunde.

3.2.2 Benutzerfreundlichkeit

NFA-04: Intuitive Bedienung

Die Auswahl von Verbindungspartnern und das Trennen der aktuellen Verbindung soll über jeweils einen Klick für den Benutzer erledigt werden können.

NFA-05: Identifikation des Verbindungspartners vor dem Verbindungsaufbau

Jeder der angezeigten Schaltflächen für einen möglichen Verbindungspartner soll die IP-Adresse und den Port des Peers enthalten.

3.2.3 Zuverlässigkeit

NFA-06: Fehlertoleranz

Die Anwendung soll bei Netzwerkfehlern oder beim Abbruch der Verbindung automatisch in einen weiterhin konsistenten Zustand bleiben.

3.2.4 Skalierbarkeit

NFA-08: Dynamisches Anzeige der verbindungsbereiten Peers

Die Anzahl der Schaltflächen zur manuellen Verbindung mit anderen Peers soll nicht durch die Fenstergröße begrenzt sein. Das bedeutet, neue Schaltflächen sollen zur Übersicht hinzugefügt werden, jedoch soll das Fenster nicht linear mit der Anzahl der anzuzeigenden Buttons größer werden.

4 Theoretische Grundlagen

4.1 Peer-Discovery in lokalen Netzwerken

Damit zwei Instanzen einer dezentralen Anwendung miteinander kommunizieren können, müssen sie sich zunächst gegenseitig finden. In zentralisierten Architekturen übernimmt ein bekannter Server diese Vermittlung; beide Teilnehmer kennen seine Adresse und registrieren sich dort. In dezentralen Systemen existiert eine solche zentrale Instanz nicht. [Bengel, 2014]

Für die Erkennung von Peers im lokalen Netzwerk existieren grundsätzlich zwei Ansätze: passive und aktive Discovery. Bei passiver Discovery annoncieren Teilnehmer ihre Präsenz über Broadcast- oder Multicast-Nachrichten (z. B. mDNS, SSDP). Jeder Teilnehmer lauscht auf einem vereinbarten Port und reagiert auf eingehende Ankündigungen. Bei aktiver Discovery hingegen durchsucht ein Teilnehmer das Netzwerk gezielt nach anderen Instanzen, indem er definierte Adressen und Ports systematisch kontaktiert. [Zisler, 2012, vgl. S. 69 ff.]

Beide Verfahren haben Implikationen: Passive Discovery erzeugt periodischen Broadcast-Traffic, wird aber von manchen Netzwerkkonfigurationen unterdrückt. Aktive Discovery erzeugt gezielten, aber potenziell umfangreichen Unicast-Traffic, funktioniert jedoch unabhängig von Broadcast-Fähigkeiten des Netzwerks.

Die in dieser Arbeit entwickelte Lösung verwendet aktive Discovery. Der Grund liegt in der Zielumgebung: In einem /24-Subnetz mit bekanntem Portbereich ist die Anzahl der zu prüfenden Endpunkte überschaubar (maximal $254 \times n_{\text{Ports}}$ Kombinationen), und die Prüfung lässt sich durch Nebenläufigkeit parallelisieren, sodass der Scanvorgang in akzeptabler Zeit abgeschlossen wird. [Zisler, 2012, vgl. S. 84 ff.]

4.2 Anwendungsschicht-Handshake als Endpunktvalidierung

Ein offener TCP-Port allein ist kein hinreichendes Kriterium dafür, dass hinter diesem Port tatsächlich die gesuchte Anwendung läuft. Auf dem gleichen Portbereich können beliebige andere Dienste aktiv sein. Eine reine Verbindbarkeit auf Transportebene (also das erfolgreiche Durchlaufen des TCP-Dreiwege-Handshakes) beweist lediglich, dass ein Prozess den Port gebunden hat und Verbindungen annimmt. [Maurer, 2019, vgl. S. 294 ff.]

Um die Identität des Gegenübers auf Anwendungsebene sicherzustellen, wird in dieser Arbeit ein eigens entworfener Handshake auf der Anwendungsschicht eingesetzt. Das Prinzip ist bewusst einfach gehalten:

1. Der anfragende Peer öffnet eine kurzlebige TCP-Verbindung zum Ziel.
2. Er sendet eine definierte Kennung (DISCOVER_SYN).
3. Nur eine Instanz der eigenen Anwendung erkennt diese Kennung und antwortet mit DISCOVER_ACK.
4. Jede andere Antwort oder ein Timeout führt zum Verwerfen des Kandidaten.

Dieser Mechanismus ähnelt strukturell dem TCP-Dreiwege-Handshake (SYN → SYN-ACK → ACK), unterscheidet sich aber in wesentlichen Punkten. Der TCP-Handshake operiert auf Schicht 4 des OSI-Modells und dient der Synchronisation von Sequenznummern sowie dem Aushandeln von Verbindungsparametern zwischen zwei Transportendpunkten. [Zisler, 2012, vgl. S. 21 ff.] Der hier beschriebene Handshake hingegen operiert auf der Anwendungsschicht (Schicht 7) und verfolgt ein anderes Ziel: Er validiert nicht die Transportverbindung, sondern die *semantische Identität* des Endpunkts. Die Frage ist nicht „Kann ich diesen Port erreichen?“, sondern „Läuft dort meine Anwendung?“.

Durch die Kombination beider Handshakes (TCP auf Transportebene und anwendungsspezifisch auf Applikationsebene) ergibt sich eine zweistufige Validierung. Die erste Stufe stellt die Erreichbarkeit sicher, die zweite die Zugehörigkeit zur erwarteten Anwendung. Fehlalarme, bei denen ein fremder Dienst fälschlich als Peer erkannt wird, werden damit praktisch ausgeschlossen.

4.3 Nebenläufigkeit im Kontext der Netzwerkcommunication

Die sequentielle Prüfung aller Adress-Port-Kombinationen im Subnetz wäre bei restriktiven Timeouts zwar korrekt, aber langsam. Wenn pro Verbindungsversuch ein Timeout von beispielsweise 200 ms angesetzt wird, ergeben sich bei 254×100 Kombinationen rund 85 Minuten Scanzeit; das ist offensichtlich nicht praxistauglich.

Die Lösung liegt in der nebenläufigen Ausführung der Prüfungen. Nebenläufigkeit bedeutet, dass mehrere Abläufe unabhängig voneinander bearbeitet werden können. [Maurer, 2019, vgl. S. 3 ff] Ob sie tatsächlich gleichzeitig (parallel) oder durch Zeitscheiben verschränkt ausgeführt werden, hängt von der verfügbaren Hardware ab; dies ist jedoch für die Korrektheit des Verfahrens unerheblich.

Go unterstützt Nebenläufigkeit als Sprachkonzept durch Goroutinen und Channels. [Maurer, 2019, vgl. S. 54 ff.] Goroutinen sind leichtgewichtige Ausführungseinheiten, die vom Go-Laufzeitsystem auf Betriebssystem-Threads verteilt werden. Channels ermöglichen die typischere Kommunikation zwischen Goroutinen ohne explizite Sperrmechanismen. Für den Discovery-Prozess bedeutet das: Jede Adress-Port-Kombination kann als eigenständige Goroutine geprüft werden, und die Ergebnisse werden über einen Channel gesammelt. Die effektive Scanzeit reduziert sich damit auf die Dauer des langsamsten Einzelversuchs plus Verwaltungsaufwand, statt auf die Summe aller Einzelversuche.

5 Entwurf

5.1 Strukturelle Veränderungen im Backend

Die wesentliche Neuerung im Backend-Entwurf ist die Einführung einer expliziten Zustandsverwaltung für Netzwerkverbindungen. Mit dem neuen Modell wird ein striktes State-Management implementiert und es wird verzichtet auf den bisherigen direkten Verbindungsaufbau. Ein zentraler `PeerManager` verwaltet den Status aller externen Teilnehmer. Diese Änderung ersetzt die sofortige Session-Etablierung durch ein kontrolliertes Verfahren zur Bereitstellung von potentiellen Peers.

Ein zentraler Aspekt ist die Trennung von Erkennung und Session-Management. Ein Endpunkt durchläuft mehrere Zustände, bevor ein Datenaustausch stattfindet. Zunächst identifiziert der Discovery-Scanner einen potenziellen Peer und führt diesen als Kandidaten in einer Liste. Erst die Verifikation über den anwendungsspezifischen Handshake überführt den Kandidaten in den Status eines verifizierten Peers. Kapitel 6 beschreibt den genauen Nachrichtenablauf.

Ein nebenläufiges Zugriffskonzept schützt die Datenintegrität bei parallelen Scans. Da der Port-Scanner hunderte Goroutinen gleichzeitig initiiert, sichern Synchronisationsprimitive den Zugriff auf die Peer-Liste ab. Das Backend gibt Zustandsänderungen asynchron über Kommunikationskanäle an das Frontend weiter. Dies stellt sicher, dass das System auch bei instabilen Netzwerkbedingungen oder einer hohen Anzahl konkurrierender Peers reaktionsfähig bleibt.

5.2 Strukturelle Veränderungen im Frontend

Bisher wird im Frontend durch eine Status-LED der Verbindungsstatus durch Farbcodierung dargestellt. Durch die Farbe Grün wird der Status "verbunden" und mit Rot der Status "nicht verbunden" signalisiert für den Benutzer. Dies wird erweitert durch die Implementierung einer reaktiven Übersicht über die verfügbaren und verbindungsreifen Instanzen im lokalen Netzwerk inklusive der Anzeige der IP-Adresse und des Ports zur Identifikation (siehe 3.1.2, 3.2.2 und 3.2.2).

Das bestehende Layout basiert auf einer hierarchischen Container-Struktur mit den zwei Bereichen: der obere Container in Form der Titelleiste mit Überschrift, Verbindungsstatus und dem Dark-Mode-Button; der untere Container mit der Visualisierung der Queue-States für Fast-, Dynamic- und Slow-Messages. Diesen beiden Bereichen wird ein dritter darunter liegender Bereich hinzugefügt, der einen scrollbaren Container mit Grid-Layout enthält und in dem dynamisch für jeden verbindungs-

dungsbereiten Peer ein Button erzeugt oder auch entfernt wird je nach Verbindungsbereitschaft (siehe 3.1.2).

Die Verwaltung der angezeigten Buttons für die verfügbaren Peers wird über eine map-basierte Struktur. Ein entdeckter Peer wird in dieser Struktur als Key-Value-Paar hinzugefügt. Als Key wird zur Identifikation die Kombination aus IP-Adresse und Port verwendet und als Value die Referenz auf den zugehörig erstellen Button zum Verbinden. Bei einem Zustandsupdate wie zum Beispiel einem neu erkannten Peer im lokalen Netzwerk wird die erhaltene Liste der erkannten Peers vom Backend mit der Map-Struktur verglichen und gegebenenfalls angepasst. Ein neu hinzugekommener Peer wird durch einen neuen Button im Frontend angezeigt und dem Grid-Layout hinzugefügt (siehe 3.1.2). Ebenfalls werden Peers, die nicht mehr erreichbar sind aus der Map-Struktur entfernt und damit auch aus dem Grid-Layout. Diese Zustandsaktualisierungen werden in Intervallen von 100ms ausgeführt (siehe 3.2.1) über eine Erweiterung des Message-Channels mit der Lister erkannter verfügbarer Peers. So wird eine synchrone Darstellung des laufenden Scan-Prozesses vom Backend im Frontend sichergestellt.

Bei erfolgreicher Verbindung mit einem erkannten Peer wird der Disconnect-Button in der Titelleiste angezeigt und gibt dem Benutzer die Möglichkeit die vorhandene Verbindung wieder zu trennen, wie es in Anforderung 3.1.2 gefordert wird. Das Grid-Layout aus Buttons verfügbarer Verbindungspartner wird zeitgleich ausgeblendet (siehe 3.1.2). Wird die Verbindung zu einem Peer getrennt wird der Disconnect-Button wieder ausgeblendet, die Verbindungsstatus-LED wird wieder rot und verfügbare Peers werden im Grid-Layout angezeigt. Durch diese gegenseitige Exklusivität der UI-Elemente wird intuitiv der aktuelle Anwendungszustand dem Benutzer signalisiert und es werden widersprüchliche Aktionen des Benutzers vermieden.

Für die Verbindungsverwaltung wird zwischen dem Backend und Frontend ein bidirektionaler Signal-Channel implementiert neben dem schon vorhandenen Message-Channel. Wird ein Button in der Peer-Übersicht im Grid-Layout angeklickt durch den Benutzer, wird die Kombination aus IP-Adresse und Port an das Backend für eine Verbindungsanfrage geschickt. Der Disconnect-Button sendet dann analog dazu ein "disconnect"-String über den Signal-Channel an das Backend für einen Graceful-Disconnect (siehe 3.1.3).

Im folgenden Kapitel 6 wird die detaillierte Implementierung der Erweiterungen an der Anwendung beschrieben.

6 Algorithmen

6.1 Implementierungsübersicht

Die implementatorische Trennung erfolgt in eine darstellende Komponente, die eine grafische Benutzeroberfläche bereitstellt, und eine serverseitige Komponente, die die Netzwerkläufigkeit und Zustandsverwaltung verantwortet. Die Kommunikation zwischen beiden Komponenten erfolgt über einen periodisch versendeten JSON-Zustand (QueueState) sowie über Steuerkanäle zur Einleitung von Verbindungsversuchen.

6.2 Portreservierungsalgorithmus

Beim Start des Programms wird ein verfügbarer TCP-Port im Bereich 50500–50600 gesucht und für die Laufzeit reserviert. Die im Quellcode vorhandene Funktion `FindAvailablePort(startPort, endPort)` prüft die Ports linear (aufsteigend) und versucht für jeden Port, einen Listener zu binden. Der erste Port, bei dem das Binden gelingt, wird als verfügbar erkannt und seine Portnummer zurückgegeben; dieser Port wird vom Programm als eigener Laufzeitport verwendet.

```
1 func FindAvailablePort(startPort, endPort int) int {  
2     for port := startPort; port < endPort; port++ {  
3         listener, err := net.Listen("tcp", fmt.Sprintf(":%d", port))  
4         if err == nil {  
5             listener.Close()  
6             return port  
7         }  
8     }  
9     return endPort  
10 }
```

Codeausschnitt 6.1: Portreservierung (Code wie im Repository)

6.3 Discovery-Protokoll

Die Peer-Erkennung basiert auf einem aktiven Scan über einen vordefinierten Portbereich (Standard: 50500–50600). Für jeden Port werden Prüfungen sowohl lokal (loopback) als auch im /24-Subnetz durchgeführt. Die zentrale Prüfroutine ist in `utils.IsPeerDiscoverable` implementiert und folgt dem Ablauf: Aufbau einer kurzlebigen TCP-Verbindung mit zeitlicher Begrenzung (Timeout), Senden einer kurzen Anwendungsschicht-Nachricht `DISCOVER_SYN` und Auswertung der

erwarteten Antwort `DISCOVER_ACK`. Dieser Anwendungsschicht-Handshake wurde gewählt, weil er die Verlässlichkeit der Erkennung erhöht (die Antwort `DISCOVER_ACK` bestätigt explizit die Präsenz der erwarteten Anwendung und reduziert damit Fehlalarme), weil durch kurze Schreib-/Leseoperationen mit restriktiven Timeouts eine effiziente Validierung möglich ist, bevor aufwendigere oder ressourcenintensive Maßnahmen ausgelöst werden, und weil in Netzwerken mit Paketfiltern oder NAT application-layer-Anfragen verlässlichere Aussagen über die Erreichbarkeit der gewünschten Anwendung erlauben.

Die folgende Abbildung 1 veranschaulicht den schematischen Ablauf der Prüfroutine.

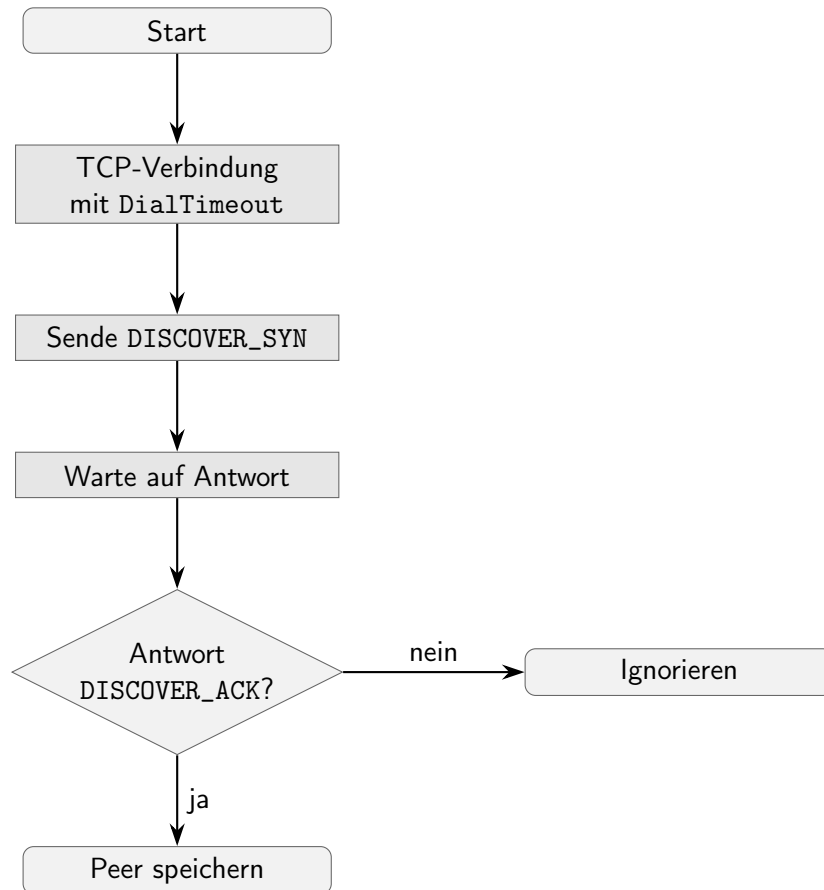


Abbildung 1: Schematischer Ablauf der Peer-Prüfung in `utils.IsPeerDiscoverable`

```

1 func DiscoverPeersSimple(serverPort int, portRange []int) []string {
2     var peers []string
3     // Produktivcode parallelisiert, hier vereinfacht
4     for _, port := range portRange {
5         if port == serverPort {
6             continue
7         }
8         addr := fmt.Sprintf("127.0.0.1:%d", port)
9         if IsPeerDiscoverable(addr) {
10             peers = append(peers, addr)
11         }
12     }
13     return peers
14 }

```

Codeausschnitt 6.2: Peer-Discovery (vereinfacht, sequentiell)

6.4 Verbindungsaufbau

Der Verbindungsaufbau erfolgt über die Nachricht `CONNECT_REQ:ip:port`. Der Nutzer wählt einen Peer aus der zuvor ermittelten Liste (siehe Discovery-Protokoll) und sendet an diesen die Verbindungsanfrage. Nach erfolgreichem Nachrichtenaustausch registrieren beide Seiten die IP-Adresse und den Port des Peers als `peerAddress` und setzen den Verbindungsstatus auf `connected = true`; damit werden weitere Verbindungsversuche unterbunden.

```

1 if strings.HasPrefix(message, "CONNECT_REQ") {
2     if connected == true {
3         return // bereits verbunden
4     }
5
6     parts := strings.Split(message, ":") // CONNECT_REQ:ip:port
7     address := parts[1]
8     port := parts[2]
9     peerAddress = fmt.Sprintf("%s:%s", address, port)
10    connected = true
11
12    conn.Write([]byte("CONNECT_OK"))
13    utils.LogInfo("Connected to peer: " + peerAddress)
14    return
15 }

```

Codeausschnitt 6.3: Vereinfachte Behandlung der `CONNECT_REQ`-Nachricht (Ausschnitt aus `handleConnection`)

Bei erfolgreicher Bearbeitung setzt der Server das Feld `peerAddress` und antwortet mit `CONNECT_OK`, woraufhin der Initiator den Verbindungsstatus auf `connected` setzt und die Discovery-Phase deaktiviert wird.

6.5 Backend - Frontend Kommunikation

Die vom Backend erfassten Zustandsdaten werden mittels Go-Kanälen an die darstellende Komponente übermittelt. Ein Go-Channel ist ein typisierter Kommunikationskanal zwischen Goroutinen; in dieser Implementierung werden unpufferte Channels benutzt, d. h. ein Sendevorgang blockiert solange, bis ein Empfänger die Nachricht liest. Das Backend serialisiert periodisch die Struktur `QueueState` als JSON-String und sendet diesen über den Nachrichtenkanal an das Frontend; das Frontend konsumiert die eingehenden Nachrichten, deserialisiert sie und aktualisiert die Anzeige. Steuerbefehle des Frontends (etwa ein Verbindungswunsch in Form von `IP:PORT`) werden über einen separaten Steuerkanal an das Backend übermittelt und dort verarbeitet.

```

1 func sendQueueStatePeriodically(messageChan chan string, interval time.Duration) {
2     for {
3         time.Sleep(interval)
4         state := QueueState{/* FastQueue, DynamicQueue, SlowQueue, Connected,
5                               DiscoveredPeers */}
6         data, _ := json.Marshal(state)
7         messageChan <- string(data)
8     }
9 }

```

Codeausschnitt 6.4: Backend: Periodisches Senden des `QueueState` (vereinfacht)

```

1 func messageHandler() {
2     for msg := range messageChannel {
3         var state QueueState
4         _ = json.Unmarshal([]byte(msg), &state)
5         fyne.Do(func() { updateUI(state) })
6     }
7 }

```

Codeausschnitt 6.5: Frontend: Empfang und Verarbeiten der Zustandsnachrichten (vereinfacht)

```

1 func sendConnectSignalToBackend(address string) {
2     signalChan <- address // IP:PORT
3 }

```

Codeausschnitt 6.6: Frontend: Senden eines Verbindungswunsches (vereinfacht)

```

1 go func() {
2     for data := range sgnChan {
3         if utils.TryToConnectToPeer(data, serverPort) {
4             peerAddress = data
5             connected = true
6         }
7     }
8 }()

```

Codeausschnitt 6.7: Backend: Empfang von Steuer-Signalen (vereinfacht)

6.6 Button-Grid Layout für Peer-Auswahl

Wie in /reffrontend-veraenderungen und 3.1.2 erwähnt, wird als zentrale Erweiterung ein Grid-Layout dem Frontend hinzugefügt zur dynamischen Anzeige verbindungsbereiter Peers. In der alten Version der Anwendung wird dem Benutzer keine Übersicht für verfügbare Peers angezeigt. Mithilfe der Verwendung von einem GridWrapLayout, wird dies geändert und innerhalb des GridWrapLayout wird pro Peer ein Button hinzugefügt und insgesamt werden alle zusammen als Raster angeordnet. Im folgenden Code-Block 6.6 wird die Initialisierung des Grids dargestellt:

```

1 peersButtonsContainer = container.New(
2     layout.NewGridWrapLayout(fyne.NewSize(200, 40))
3 )
4 peersButtonsMap = make(map[string]*widget.Button)

```

Codeausschnitt 6.8: Initialisierung des Button-Grid Containers

Mithilfe des GridWrapLayout werden die Buttons mit einer festgelegten Größe von 200x40 Pixel gerendert und automatisch mit Umbrüchen angeordnet, falls die aktuelle Breite des Containers nicht ausreicht. So wird eine responsive Darstellung der Buttons ermöglicht, bei der die Buttons angepasst an die Größe des Anwendungsfensters dem Nutzer bereitgestellt werden. Wie in 6.6 implementiert, wird der GridWrapLayout-Container in einen Scroll-Container mit einer Höhe von 150 Pixel umhüllt, damit die Fenstergröße nicht linear steigt bis in unendliche Größen wie 3.2.4 beschrieben.


```

1 peersScroll := container.NewVScroll(peersButtonsContainer)
2 peersScroll.SetMinSize(fyne.NewSize(0, 150))

```

Codeausschnitt 6.9: Einbettung in Scroll-Container

Jeder der gerenderten Buttons zur Darstellung der verbindungsreifen Peers wird mit einem Computer-Icon und der Importance-Stufe `HighImportance` erstellt. Somit wird dieser visuell hervorgehoben durch die blaue Farbe und den Kontrast im Standard-Theme. Mit der Callback-Funktion jedes Buttons wird die jeweilige Kombination aus IP-ADRESSE:PORT über den Signal-Channel `signalChan` an das Backend geschickt. Diese Neuerungen werden implementiert im unteren Codeblock 6.7.

6.7 Dynamische Peer-Button-Verwaltung

Im Frontend wird eine dynamischen Verwaltung der Peer-Buttons im `GridWrapLayout` hinzugefügt, basierend auf der `DiscoveredPeers`-Liste vom Backend aus dem Message-Channel (siehe 3.1.1). Wie in 3.1.2 beschrieben, werden nur im nicht verbundenen Zustand neue Peers, die vom Backend erkannt werden, zum Grid-Layout der Peer-Übersicht hinzugefügt und angezeigt. Außerdem werden verschwundene Peers wieder entfernt, damit diese nicht mehr im Grid-Layout angezeigt werden. Die schon im Teilkapitel 5.2 angesprochene Map-Struktur `peersButtonsMap` wird verwendet, um eine Verbindung zwischen dem Button in Form einer Referenz und der Kombination IP-ADRESSE:PORT zu erschaffen. Durch den Text im Button wird im Format IP-ADRESSE:PORT die IP-Adresse und der Port des Peers nach 3.2.2 visualisiert. Diese Umsetzungen können dem folgenden Code-Block 6.7 entnommen werden.

```

1  if !state.Connected {
2      // Neue Peers hinzufügen
3      for _, peer := range state.DiscoveredPeers {
4          if _, exists := peersButtonsMap[peer]; !exists {
5              peerCopy := peer
6              btn := widget.NewButtonWithIcon(" " + peerCopy, theme.ComputerIcon(),
7                  func() { sendConnectSignalToBackend(peerCopy) })
8              btn.Importance = widget.HighImportance
9              peersButtonsMap[peerCopy] = btn
10             peersButtonsContainer.Add(btn)
11         }
12     }
13
14     // Verschwundene Peers entfernen
15     for addr, btn := range peersButtonsMap {
16         found := false
17         for _, p := range state.DiscoveredPeers {
18             if p == addr { found = true; break }

```

```

19     }
20     if !found {
21         peersButtonsContainer.Remove(btn)
22         delete(peersButtonsMap, addr)
23     }
24 }
25 }

```

Codeausschnitt 6.10: Dynamische Button-Verwaltung (Ausschnitt aus updateUI)

Ein Button-Closure wird mit der Variable `peerCopy` verhindert, was ein in Go häufig auftretendes Problem darstellt, bei dem alle Closures auf dieselbe Variable in der Schleife referenzieren würden. Mit der Erstellung einer Kopie wird die Eindeutigkeit der Variable und das korrekte Anzeigen von IP-Adresse und Port im Button-Text sichergestellt.

6.8 Zustandsabhängige UI-Sichtbarkeit

Die Zustandabhängigkeit der UI wird mit 3.1.2 gefordert. Das Button-Grid-Layout wird im verbundenen Zustand ausgeblendet, damit keine zweite Verbindungsanfrage gesendet werden könnte nach Anforderung 3.1.2 und der Disconnect-Button wird eingeblendet. Im nicht verbundenen Zustand wird dieses Verhalten umgekehrt, das Button-Grid-Layout wird dargestellt und der Disconnect-Button wird temporär entfernt. Dieses Verhalten wird wie in 6.8 implementiert mithilfe der Abfrage der Variable `connected` und den Container-spezifischen Methoden `.Hide()` und `.Show()`.

```

1 func updateUI(state QueueState) {
2     connected = state.Connected
3
4     if connected {
5         disconnectButton.Show()
6         peersButtonsContainer.Hide()
7         // Alle Peer-Buttons entfernen
8         for addr, btn := range peersButtonsMap {
9             peersButtonsContainer.Remove(btn)
10            delete(peersButtonsMap, addr)
11        }
12    } else {
13        disconnectButton.Hide()
14        peersButtonsContainer.Show()
15    }
16
17    updateLEDColor(connected)
18 }

```

Codeausschnitt 6.11: Zustandsabhängige UI-Aktualisierung (Ausschnitt)

Durch den gegenseitige Ausschluss der Darstellung von Disconnect-Button und Button-Grid-Layout werden widersprüchliche Nutzerinteraktionen verhindert und es wird visuelle Eindeutigkeit sichergestellt. In Codeblock 6.8 wird mit der vollständigen Entfernung aller Buttons, während eine Verbindung erfolgreich hergestellt wird, wird garantiert, dass keine veralteten Buttons von nicht mehr verfügbaren Peers als Artefakte noch im Frontend erhalten bleiben.

6.9 Sicherheits- und Netzwerkeffekte

Das aktive Scanning erzeugt messbaren Netzwerktraffic; in restriktiven Umgebungen kann dies zu Rate-Limiting oder zu Sicherheitswarnungen führen. Die Implementierung ist daher ausschließlich für lokale Netzwerke konzipiert.

7 Diskussion

Die neue Implementierung ermöglicht explizite Peer-Auswahl statt adhoc-Verbindungen. Dies hat unmittelbare Folgen für die praktische Anwendbarkeit.

7.0.1 Alte Version: Ohne explizite Peer-Auswahl

Die Abbildung 2 zeigt die ursprüngliche Oberflächengestaltung. Die Anwendung konnte keine Liste erreichbarer Peers darstellen. Der Benutzer hatte keinen Überblick über, welche Instanzen im Netzwerk verfügbar waren. Eine Verbindung musste durch manuelle Eingabe der Zieladresse erfolgen oder das System stellte automatisch die erste verifizierte Verbindung her. Dies führte zu einer reaktiven Herangehensweise: Der Nutzer wartete, bis sich eine Verbindung aufbaute, konnte aber nicht aktiv zwischen mehreren Kandidaten wählen.

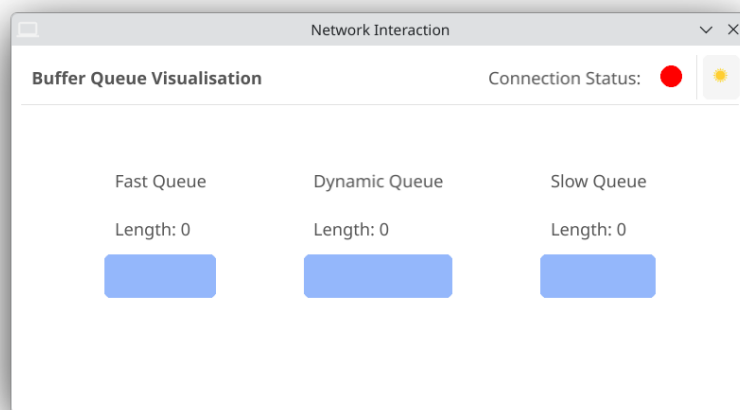


Abbildung 2: Alte Version: Fehlende Peer-Discovery-Visualisierung

7.0.2 Neue Version: Mit Peer-Auswahlliste

Abbildung 3 demonstriert die neue Oberflächengestaltung. Die Anwendung zeigt nun eine Echtzeit-Liste aller im lokalen Netzwerk erkannten Peer-Instanzen. Der Nutzer sieht: Anzahl der verfügbaren Kandidaten, deren Netzwerkadressen und kann über die Buttons gezielt eine Verbindung zu einem ausgewählten Peer aufbauen. Diese proaktive Herangehensweise gibt dem Benutzer volle Kontrolle über die Verbindungsentscheidung.

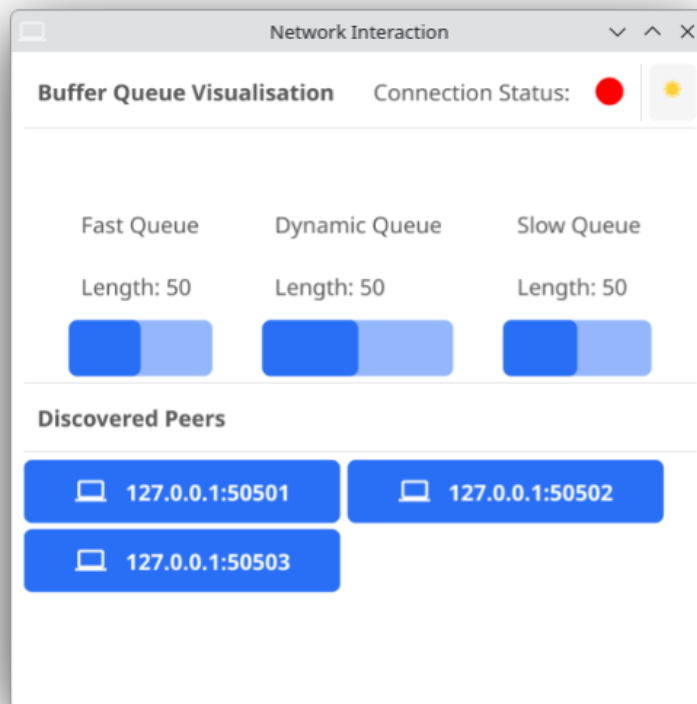


Abbildung 3: Neue Version: Explizite Peer-Discovery-Liste mit Auswahlmöglichkeit

7.1 Auswirkungen der neuen Peer-Auswahl

Der Unterschied liegt in der Kontrolle. Die alte Version machte verfügbare Peers unsichtbar; eine Verbindung entstand automatisch zur ersten erkannten Instanz. Der Nutzer hatte keinen Überblick und konnte nicht wählen.

Die neue Version zeigt alle erkannten Instanzen in einer Live-Liste mit ihren Netzwerkadressen. Der Benutzer sieht sofort, wie viele Kandidaten verfügbar sind, und kann gezielt auswählen, zu welcher Instanz die Verbindung aufgebaut wird. Dies ist entscheidend, wenn mehrere Instanzen der Anwendung im gleichen Netzwerk laufen (etwa weil eine versehentlich nicht geschlossen wurde oder weil mehrere Nutzer parallel aktiv sind). In der alten Variante war dies ein Problem: Das System hätte blind zur ersten Instanz verbunden, ohne dem Nutzer eine Wahl zu geben. Mit der neuen Implementierung wird die dezentrale Architektur vollständig umgesetzt. Jeder Nutzer kann bewusst mit jeder anderen Instanz kommunizieren, unabhängig davon, wie viele parallel aktiv sind. Das verdeutlicht das Kernkonzept: kein zentraler Server existiert und jeder Teilnehmer agiert gleichberechtigt.

8 Fazit

Im Rahmen dieser Studienarbeit II wird die in Studienarbeit I entwickelte Desktop-Anwendung erfolgreich erweitert um ein interaktives Verbindungsmanagementsystem. Mithilfe der implementierten Erweiterungen wird es Nutzern der Anwendung ermöglicht, manuell einen Verbindungspartner aus einer visuell veranschaulichten Liste von anderen verfügbaren Peers im lokalen Netzwerk auswählen zu können. Von der Entwicklung werden sowohl strukturelle Anpassungen im Backend als auch die Erweiterung des Frontends um ein dynamisches Peer-Auswahlsystem abgedeckt.

Die funktionalen und nicht-funktionalen Anforderungen aus Kapitel 3 werden durch die Implementierung vollständig umgesetzt. Im Backend wird eine kontinuierliche Peer-Discovery im nicht verbundenen Zustand (FA-01 FA-02) implementiert und im Frontend eine Übersicht zur Visualisierung verbindungsbereiter Peers in Form eines Grids mit Buttons (FA-04, FA-04, NFA-08). Durch die zusätzlichen Implementierungen im Backend, wie die Umsetzung des Verbindungs-Handshake-Protokolls (FA-05), den bidirektionalen Informationsaustausch (FA-06) und die Verbindungsexklusivität (FA-07), wird eine Verbindungsverwaltung innerhalb der Anwendung erschaffen. Zur manuellen Verbindungstrennung wird über die Umsetzung manueller Disconnects durch einen Button im Frontend (FA-08), Graceful Disconnects (FA-09) und eine Zustandssynchronisation bei Disconnects (FA-10) ermöglicht. Darüber hinaus werden sowohl im Frontend als auch im Backend der Anwendung nicht funktionale Anforderungen umgesetzt hinsichtlich der Performance (NFA-01 bis NFA-03), Benutzerfreundlichkeit (NFA-04 NFA-05), Zuverlässigkeit (NFA-06) und Skalierbarkeit (NFA-08).

Für zukünftige Erweiterungen des Projekts bestehen verschiedene Möglichkeiten. Eine kontinuierliche Peer-Discovery auch während bestehender Verbindungen würde es ermöglichen, ohne Trennung der aktuellen Verbindung bereits nach potenziellen neuen Verbindungspartnern zu suchen. Die Implementierung eines Verbindungsverlaufs könnte dem Nutzer Einblick in vergangene Verbindungen geben. Zusätzlich könnte ein Mechanismus zur Priorisierung bevorzugter Peers implementiert werden, etwa durch das Speichern häufig genutzter Verbindungen. Eine weitere sinnvolle Erweiterung wäre die Unterstützung mehrerer gleichzeitiger Verbindungen, wobei die aktuelle Exklusivität (eine Verbindung pro Instanz) aufgehoben werden müsste.

Durch die Fertigstellung dieser Arbeit wurde die ursprüngliche Anwendung um wesentliche Funktionalitäten erweitert, die die Nutzerfreundlichkeit und Kontrolle über Netzwerkverbindungen deutlich verbessern. Die klare Architektur mit Trennung zwischen automatischer Erkennung und manueller Steuerung schafft die Grundlage für weitere Entwicklungen im Bereich dezentraler Peer-to-Peer-Anwendungen.

Literaturverzeichnis

- [Bengel, 2014] Bengel, G. (2014). *Grundkurs Verteilte Systeme*. Springer Vieweg.
- [Docs, 2025a] Docs, G. G. (2025a). component package - gioui.org/x/component - Go Packages — pkg.go.dev. <https://pkg.go.dev/gioui.org/x/component>. [Accessed 15-08-2025].
- [Docs, 2025b] Docs, G. G. (2025b). fyne package - fyne.io/fyne/v2 - Go Packages — pkg.go.dev. <https://pkg.go.dev/fyne.io/fyne/v2>. [Accessed 15-08-2025].
- [Docs, 2025c] Docs, G. G. (2025c). layout package - gioui.org/layout - Go Packages — pkg.go.dev. <https://pkg.go.dev/gioui.org/layout>. [Accessed 15-08-2025].
- [Docs, 2025d] Docs, G. G. (2025d). net package - net - Go Packages — pkg.go.dev. <https://pkg.go.dev/net>. [Accessed 15-08-2025].
- [Docs, 2025e] Docs, G. G. (2025e). net package - net - Go Packages — pkg.go.dev. <https://pkg.go.dev/net#Conn>. [Accessed 15-08-2025].
- [Docs, 2025f] Docs, G. G. (2025f). rand package - math/rand - Go Packages — pkg.go.dev. <https://pkg.go.dev/math/rand>. [Accessed 15-08-2025].
- [Docs, 2025g] Docs, G. G. (2025g). sync package - sync - Go Packages — pkg.go.dev. <https://pkg.go.dev/sync>. [Accessed 15-08-2025].
- [Docs, 2025h] Docs, G. G. (2025h). tcp package - gvisor.dev/gvisor/pkg/tcpip/transport/tcp - Go Packages — pkg.go.dev. <https://pkg.go.dev/gvisor.dev/gvisor/pkg/tcpip/transport/tcp>. [Accessed 15-08-2025].
- [Docs, 2025i] Docs, G. G. (2025i). The Go Programming Language Specification - The Go Programming Language — go.dev. <https://go.dev/ref/spec>. [Accessed 15-08-2025].
- [IBM, 2024] IBM (2024). What Is Network Attached Storage (NAS)? | IBM — ibm.com. <https://www.ibm.com/think/topics/network-attached-storage>. [Accessed 12-08-2025].
- [Kaufmann and Mülder, 2022] Kaufmann, J. und Mülder, W. (2022). *Grundkurs Wirtschaftsinformatik*. Springer Vieweg.
- [Mandl, 2024] Mandl, P. (2024). *TCP, UDP und QUIC Internals*. Springer Vieweg.

- [Maurer, 2019] Maurer, C. (2019). *Nichtsequentielle und Verteilte Programmierung mit Go: Synchronisation nebenläufiger Prozesse: Kommunikation – Kooperation – Konkurrenz*. Computer Science and Engineering (German Language). Springer Fachmedien Wiesbaden.
- [Wagenknecht, 2004] Wagenknecht, C. (2004). *Programmierparadigmen*. Vieweg+Teubner Verlag.
- [Zisler, 2012] Zisler, H. (2012). *Computer-Netzwerke: Grundlagen, Funktionsweise, Anwendung ; [Theorie und Praxis: von der MAC-Adresse bis zum Router ; TCP/IP, IPv4, IPv6, (W)LAN, VPN, VLAN u.v.m. ; Konfirgration, Planung, Aufbau und sicherer Betrieb von Netzwerken]*. Galileo Computing. Galileo Press.

KI-Verzeichnis

KI-basiertes Hilfsmittel	Einsatzform	Betroffene Teile der Arbeit
Generative KI (Claude)	Komprimierung von Textpassagen	Kapitel 8Fazit
DeepL Translator	Übersetzung von Textpassagen	Kapitel Theoretische Grundlagen
Generative KI (Claude)	Einstieg in Themen fürs Verständnis	Kapitel Theoretische Grundlagen

Ehrenwörtliche Erklärungen

Hiermit versichere ich, dass ich die vorliegende Arbeit in allen Teilen selbstständig angefertigt und keine anderen, als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt habe. Sämtliche wörtlichen oder sinngemäßen Übernahmen und Zitate, sowie alle Abschnitte, die mithilfe von KI-basierten Tools entworfen, verfasst und/oder bearbeitet wurden, sind kenntlich gemacht und nachgewiesen.

Im Anhang meiner Arbeit habe ich sämtliche KI-basierte Hilfsmittel angegeben. Diese sind mit Produktnamen und formulierten Eingaben (Prompts) in einem KI-Verzeichnis ausgewiesen.

Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstöße gegen das Datenschutz- und Urheberrecht oder Plagiate.

Berlin, 15.02.2026

Ort, Datum

A. Betke

Alexander Betke

Berlin, 15.02.2026

Ort, Datum

T. Leuthardt

Theo Leuthardt