Verteilte Systeme

VS Praktikum SoSe 2025

Manh-An David Dao, Philipp Patt, Jannik Schön, Marc Siekmann

17. Juli 2025

Inhaltsverzeichnis

1 Einführung und Ziele

Diese Dokumentation beschreibt Entwurf und Kontext einer Middleware, die im Rahmen des Praktikums „Verteilte Systeme SoSe 2025“ entwickelt wird. Ziel des Praktikums ist der Entwurf und die Implementierung eines verteilten Steuerungssystems für Roboterarme. Die Middleware bildet dabei eine zentrale Komponente, die die Komplexität der Verteilung für die Anwendungsebene abstrahieren soll.

Dieses Dokument ist auf Basis des Skriptes von Prof. M. Becke, *Verteilte Systeme (Skript zum Modul v0.9-6)*, veröffentlicht März 19, 2025 enstanden.

1.1 Aufgabenstellung

Die Hauptaufgabe der zu entwickelnden Middleware besteht darin, die Kommunikation und Koordination zwischen den verteilten Komponenten des Steuerungssystems zu ermöglichen. Sie soll eine Abstraktionsschicht zwischen der eigentlichen Anwendungslogik (Roboterarmsteuerung) und den darunterliegenden Betriebssystem- und Netzwerkdiensten bilden. Gemäß den Prinzipien verteilter Systeme nach Tanenbaum & van Steen, die auch im zugrundeliegenden Skript referenziert werden, verfolgt die Middleware das Ziel, die Verteilung weitestgehend zu verbergen (Transparenz). Zu den Kernfunktionen, die eine Middleware typischerweise bereitstellt und die hier adressiert werden sollen, gehören:

- Zuständig für die Verteilungstransparenz
- Bereitstellung der Kommunikation mittels standardisierter Protokolle. Dies wird (asynchrone) Kommunikation-Pattern umfassen, wie z.B. Remote Procedure Calls (RPC).
- Unterstützung zur Konvertierung von Datenformaten zwischen Systemen.
- Bereitstellung von Protokollen zur Namensauflösung (Service Discovery), um Ressourcen einfach zu identifizieren und zu referenzieren.
- Mechanismen der Skalierbarkeit, wie z.B. Replikation
- Security Mechanismus, dass nicht akzeptierte Nachrichten verworfen werden.

Die Middleware dient als Vermittler, der zwischen der Applikation und der Runtime/OS Schicht Daten- und Informationsaustausch ermöglicht.

Zusammengefasst soll die Middleware Steuerung der einzelnen Nodes (Servos) eines jeden im System befindlichen Roboterarms ermöglichen. Dazu gehören die Erkennung jeder Node, und die Weiterleitung der Steuerbefehle.

1.2 Qualitätsziele

Tabelle 1.1: Qualitätsziele der Software Engineering

Ziel	Beschreibung	Metrik
Zuverlässigkeit	Fehler dürfen den Betrieb nicht gefährden. Fehlererkennung und -toleranz müssen integriert sein.	Das System ist über dem gesamten Abnahmezeitraum stabil (ca. 1,5 h). Definiert aufgetretene Fehler werden kommuniziert.
Skalierbarkeit	Zusätzliche Nodes oder Komponenten sollen ohne Änderungen an der bestehenden Architektur integrierbar sein.	Es können bis zu 253 Nodes hinzugefügt und entfernt werden
Wartbarkeit	Der Code muss übersichtlich sein, gut dokumentiert sein und wenig Komplexität enthalten.	Zyklomatische Komplexität ≤ 10 und LOC ≤ 30 pro Methode/Funktion exklusive Kommentar
Ressourcenteilung	Alle dem Netzwerk hinzugefügten Nodes können sich registrieren und anschließend miteinander kommunizieren	Das Namensregister ist im gesamten Netzwerk verfügbar
Offenheit	Zugänglichkeit Interoperabilität Portabilität	Einzelne Softwarekomponenten können, ausgetauscht oder portiert werden, um zb Standards bzgl. Kommunikation und Datenstruktur zu tauschen oder um die Plattform zu wechseln. Die Funktionalität bleibt gleich
Zugriffstransparenz	Die Umsetzung Kommunikation zwischen den Nodes ist für den Benutzer nicht erkennbar	Die Applikation kommuniziert über Namen. Die eigentliche Kommunikation bleibt versteckt.

Lokalitäts-Transparenz	Die Netzwerk- und Softwarestruktur ist nach außen unsichtbar	Das Interface nach außen ist einheitlich und verschleiert die Implementierung

1.3 Stakeholder

Tabelle 1.2: Interessen der Stakeholder

Stakeholder	**Interesse**
Betreiber	• Portabilität: Das System kann auf verschiedenen Plattformen betrieben werden. • Zuverlässigkeit: Die Middleware kann über den gesamten benötigten Zeitraum ohne Ausfälle genutzt werden
Entwicklerteam Middleware	• Wartbarkeit • Portabilität: Das System kann auf verschiedenen Plattformen betrieben werden (z.B Testen) • Austauschbarkeit: Softwaremodule können ohne großen Aufwand ersetzt werden
Entwicklerteam Applikation	• Middleware bietet vollständige Funktionalität • Middleware-Schnittstellen sind vollständig beschrieben. • Zuverlässigkeit und Reaktionszeit der von der Middleware bereitgestellten Kommunikationsdienste.
Professor	• Zugang zu allen Arbeitsmitteln zwecks Bewertung und Kontrolle • Das Endprodukt besitzt alle geforderten Funktionalitäten

2 Randbedingungen

Dieses Kapitel beschreibt die Rahmenbedingungen, unter denen die Middleware entworfen und implementiert werden muss.

2.1 Technische Randbedingungen

- **Betriebsumgebung:** Die Middleware muss auf einer heterogenen Umgebung aus verschiedenen Hardware-Plattformen und Betriebssystemen laufen können. Im spezifischen Projektkontext umfasst dies mindestens ein ITS-Board (STM32F4) und mehrere Raspberry Pis.
- **Netzwerk:**
  - Die Kommunikation findet innerhalb eines begrenzten Netzwerks statt, im Projektkontext ein /24 Netzwerk. Die Middleware baut auf den grundlegenden Kommunikationsdiensten des Betriebssystems und Netzwerks auf.
  - Es wird mit IPv4 kommuniziert.
- **Hardware:** Die Middleware muss auf der Hardware laufen, die für das Steuerungssystem verwendet wird: ITS-Board und Raspberry Pi 3.
  - ITS-Board (STM32 Nucleo-144 Board):
    - OS: Kein Betriebssystem vorhanden (RTOS möglich)
    - CPU: STM32F4
  - Raspberry Pi 3
    - OS: Raspian GNU/Linux 9
    - CPU: ARMv7 Processor rev 5
    - RAM: 927 MB
- **Anbindung:** Die Middleware muss in der Lage sein, mit der Anwendungsschicht und den System-/Netzwerkschichten zu interagieren.
- **Sprachen:** Die Middleware muss in den Sprachen C und JAVA umgesetzt werden.

2.2 Organisatorische Randbedingungen

- **Zeit:** Entwicklungszeitraum beträgt 12 Wochen.
- **Vorwissen:** Einige Konzepte und Herangehensweisen werden erst im Laufe der 12 Wochen gelernt.
- **Budget:** Es steht kein Budget zur Verfügung.

3 Kontextabgrenzung

3.1 Fachlicher Kontext

3.1.1 Akteure und Rollen

Akteur / Rolle	**Beschreibung**
Server (Applikation)	Ein einzelner Teilnehmer. Meldet sich eigenständig bei der Middleware mit seinen Diensten an.
Client (Applikation)	Ein einzelner Teilnehmer. Kann die Dienste der anderen Teilnehmern aufrufen.
Health-Observable (Applikation)	Teilnehmer, der Lebenszeichen schickt
Health-Observer (Applikation)	Teilnehmer, der bei ausbleibenden Lebenszeichen benachrichtigt wird

3.1.2 Fachliche Aufgaben (Use Cases)

ID	Name	Beschreibung
U1	Informationen weiterleiten	Die Middleware leitet RPCs von der Applikation an die entsprechenden Nodes weiter.
U2	Node registrieren	Ein Server registriert seine Dienste bei der Middleware mit einem eindeutigen Identifier. Dem Identifier wird dessen Socket-Adresse zugeordnet.
U3	Namen auflösen	Die Middleware ermittelt, welche Dienste bei welcher Socket-Adresse zu finden sind.
U4	Marshalling	Die Middleware definiert eine IDL. Diese muss dafür sorgen, dass die Datenstrukturen und Parameter der Funktionsaufrufe via RPC korrekt in ein übertragbares Format (Marshalling) umgewandelt und am Zielsystem wieder entpackt (Unmarshalling) werden.
U5	Transport	Die RPCs werden via UDP über die Netzwerkschicht transportiert. Das ist durch die Echtzeitanforderung bedingt. Kommunikation findet asynchron statt. Synchronizität wird durch mehrere asynchrone RPCs umgesetzt.
U6	Sicherheit	Versendete RPCs haben (zwischen Client und Server) Echtzeitanforderungen.
U7	Watchdog	Der Watchdog informiert Subscriber ob andere Services verfügbar sind.

3.1.3 Fachliche Randbedingungen

- Es können bis zu 253 Nodes mit unterschiedlichen IPv4-Adressen gleichzeitig betrieben werden.
- Es wird IPv4 gefordert, daher muss das TCP/IP-Modell für den Transport genutzt werden.
- Die Middleware dient als Vermittler bzw, Abstraktionsschicht. Dadurch ist mindestens eine indirekte Kopplung zu implementieren. Um eine hohe Skalierbarkeit oder Fehlertoleranz zu erreichen, ist eine losgekoppelte Kopplung zu implementieren.

3.2 Technischer Kontext

Die Middleware positioniert sich technisch als Vermittlungsschicht zwischen der Anwendungsebene und den darunterliegenden Betriebssystem- und Kommunikationsdiensten. Sie ist über mehrere Teilnehmer verteilt, die über ein lokales /24 Netzwerk miteinander verbunden sind. Die Middleware nutzt Netzwerkprotokolle, insbesondere UDP, sowie Betriebssystemfunktionen,

um eine zuverlässige Kommunikation zu gewährleisten. Gleichzeitig bietet sie den Anwendungen eine abstrahierte, einheitliche Schnittstelle, die die Heterogenität der zugrundeliegenden Systeme verbirgt. Aus diesem Grund eignet sich in den folgenden Abschnitte eine funktionale Zerlegung. Mit dieser kann die Middleware alle geforderten Ziele eines verteilten Systems einhalten. Mit einer ressourcen-basierten Zerlegung ist dies komplexer.

3.2.1 Technische Anforderungen an die Middleware

Um die fachlichen Use Cases umzusetzen, muss die Middleware folgende technische Funktionen bereitstellen:

- **Informationen weiterleiten U1:**
  - Schnittstelle zur Applikation zum entfernten Funktionsaufruf und dessen Parameter.
  - Einheitliche Übersetzung und Serialisierung.
  - Namensauflösung, ggf. aus Cache.
  - Transport der Nachricht über Netzwerk.
  - Deserialisierung der empfangenen Aufrufs.
  - Schnittstelle zur Applikation, um lokalen Funktionsaufruf auszuführen.
- **Node registrieren U2:**
  - Schnittstelle zur Applikation mit Gruppen Name, Funktionsname und seiner Socket-Adresse via RPC
  - Einheitliche Übersetzung und Serialisierung
  - Senden an bekannten Namensserver
  - Deserialisierung der empfangenen Aufrufs am Namensserver
  - Eintragen der Informationen
- **Namen auflösen U3:**
  - Senden der Anfrage mit Parameter (Gruppen Name, Funktionsname, eigener Gruppenname, Funktionsname der Antwort) via RPC
  - (Einheitliche Übersetzung und Serialisierung)
  - Senden an bekannten Namensserver
  - Warten auf Antwort
  - (Deserialisierung des empfangenen Aufrufs am Namensserver)
  - Auflösen der Socket-Adresse zum Funktions- und Gruppennamen
  - Aufruf der Antwortsfunktion anhand Gruppenname mit Parameter(Socketadresse)
  - (Einheitliche Übersetzung und Serialisierung)
  - Senden an vorherigen Sender

  - (Deserialisierung der empfangenen Aufrufs)
  - Einsetzen der Socketadresse im Cache

- **Marshalling U4:**

  - Aufruf des Marshallings mit Parameter (Funktionsname, Liste von Typisierten Parameter)
  - Übersetzen der Funktion und der Parameter mittels IDL in ein serialisierbaren Typen (char-array)
  - (Versenden der Nachricht)
  - Aufruf des Unmarshallings mit Parameter (serialisierter Funktionsaufruf)
  - Übersetzen serialisierter Funktionsaufruf in eine Funktion und der Parameter (Funktionsname, Liste von Typisierten Parameter) mittels IDL

- **Transport U5:**

  - Aufruf der Netzwerkschicht mit Parameter (Serialisierter Funktionsaufruf)
  - Empfang des serialisierten Funktionsaufrufs
  - Deserialisierung der Serialisierung

- **Sicherheit U6:**

  - Auslesen des eigenes Timestamps
  - (Einheitliche Übersetzung und Serialisierung)
  - Senden an alle möglichen Sender
  - empfangen der Nachricht und Deserialisieren
  - Setzen des Timestamps bei zugehörigem Empfänger
  - Nutzung des Timestamps bei Nachricht an Empfänger

- **Watchdog U7:**

  - Senden eines Subscriber-Requests an einen zentralen Watchdog
  - Periodisches Senden eines Heartbeats an den Watchdog
  - Senden von periodischen Healthreports an die Subscriber

3.3 Externe Schnittstellen

Die Middleware besitzt folgende Schnittstellen:

- **Schnittstelle zur Anwendungsebene:** Bietet eine einheitliche API, über die Anwendungen (z. B. auf ITS-Board oder Raspberry Pis) verteilte Dienste nutzen, Nachrichten senden und empfangen können, sowie sich als Teilnehmer registrieren können, unabhängig von Netzwerkdetails oder physikalischer Verteilung.

- **Schnittstelle zu System- und Netzwerkschichten:** Nutzt Betriebssystem- und Netzwerkdienste (TCP/IP, IPv4) zur Nachrichtenübermittlung und Ressourcenverwaltung. Diese Schnittstelle ist für die Anwendungen verborgen.

4 Lösungsstrategie

Kommunikation

Die losgekoppelte Kopplung, fordert eine asynchrone Kommunikation ein. Durch die festgelegte funktionale Zerlegung ist RPC ein geeigneter Kommunikationsmechanismus. Die losgekoppelte Kopplung fordert im TCP/IP Stack das Protokoll UDP ein, da TCP als verbindungsorientiertes Protokoll durch die ACK Pakete synchron arbeitet. Registrierungsvorgänge werden während der Laufzeit gespeichert, während Steuerbefehle transient und zeitkritisch behandelt werden. Um sicherzustellen, dass pro RPC Aufruf nur ein UDP-Paket verschickt wird, darf die Payload die Größe von 256 Bytes nicht überschreiten.

Marshalling

Die Verwendung von RPC über UDP erfordert eine Serialisierung. Daher wird eine IDL zur Zuordnung und Serialisierung von Funktionsaufrufen entwickelt.

Namensauflösung

Für die Namensauflösung wird eine hierarische Struktur gewählt. Durch die funktionale Zerlegung wird jeder Funktion eine Socket-Adresse, bestehend aus IPv4 und Port, zugewiesen. Diese Funktionen gehören jeweils zu einem Prozess bzw. zu einem Softwareblock. Daraus folgt, dass die Gruppen nach dem Softwareblock, im folgenden Service genannt, bestehen. Jeder Service besteht aus einer Gruppe von Funktionen, denen jeweils eine individuelle Socket-Adresse zugeordnet ist. Bei Start muss sich also jede Funktion mit Service, Funktionsnamen und Socket-Adresse registrieren.

Fehlerbehandlung

Fehler während der Kommunikation via RPC über UDP wird toleriert. Durch die Anforderung Safety dürfen UDP Pakete, die nicht rechtzeitig ankommen (Latenz) nicht verarbeitet werden. Dies ist mit dem CAP-Theorem zu begründen, nach dem Konsistenz und Verfügbarkeit nicht gleichzeitig in einem verteilten System bestehen können. Da Konsistenz priorisiert wird, ist die zeitliche Richtigkeit der ankommenden Pakete zu erfassen.

Sicherheit / Safety

Durch die Bedingung des CAP-Theorems wird sich durch die Safety-Anforderung für die Konsistenz entschieden. Dadurch bekommt jede RPC-Nachricht, die von der Applikation ausgelöst wurde einen Timestamp, der von dem jeweiligen Empfänger zuvor an den Sender propagiert

wurde. So kann der Empfänger sowohl Reihenfolge als auch Rechtzeitigkeit bestimmen. Aus den Qualitätszielen geht hierbei eine maximale Differenz zwischen Auslösen der Funktion und Ankommen des RPC-Pakets von 250 ms hervor.

4.1 Funktionen

Aus den vorherigen Feststellungen und den Use-Cases ergeben sich folgende Funktionen.

Tabelle 4.1: Funktionsbeschreibungen RPC-Kommunikation

Funktion	Beschreibung	Vor-Nachbedingungen
void invoke(string functionName, RpcValue[] params)	**Schnittstelle zur Applikation** um RPCs an entfernte Ziele zu senden.	IDL der Middleware kann den Typen Marshallable der Applikation serialisieren
void call(string functionName, RpcValue[] args)	**Schnittstelle zur Applikation** um RPC in der lokalen Applikation aufzurufen	Die Funktion mit den Parametern muss existieren
int marshall(string functionName, RpcValue[] params, string* payload, uint32_t timestamp)	Führt Marshalling durch	Marshallable und buffer darf nicht NULL sein. Nach: return 0 oder -1
void unmarshall(string payload, string functionName, RpcValue[] params, uint32_t timestamp)	Führt Unmarshalling durch	Marshallable und buffer darf nicht NULL sein. Nach: return 0 oder -1
void setTimestamp(string servicename, string function, uint32_t timestamp)	RPC-Funktion, die einen Timestamp bei anderen Teilnehmer setzt	Vor: servicename, function nicht NULL
bool set_or_update_timestamp(string servicename, string function, int timestamp)	interne funktion, die einen Timestamp setzt wenn empfangen	Vor: servicename, function nicht NULL Nach: true, wenn erfolgreich, false, wenn nicht

Funktion	Beschreibung	Vor-Nachbedingungen
bool get_timestamp(string servicename, string function, int out_timestamp)	interne funktion, die einen Timestamp ausliest	Vor: servicename, function nicht NULL Nach: true, wenn erfolgreich, false, wenn nicht
int cache_store(string servicename, string functionname, string socket, int time);	interne funktion, die einen socket in den cache legt	Vor: servicename, function nicht NULL Nach: 0, wenn erfolgreich, -1, wenn nicht
string cache_lookup(string servicename, string functionname);	interne funktion, die einen socket aus dem cache liest	Vor: socket oder null

Tabelle 4.2: Funktionsbeschreibungen Name Service

Funktion	Beschreibung	Vor-Nachbedingungen
void register(string service, string function, string socket)	**Schnittstelle zur Applikation** für eine Funktion eines Services einer Node, um sich zu beim System zu registrieren.	Jede Node (Servo) muss seinen eigenen Identifier und Socket vor der Registrierung kennen.
string resolve(string servicename, string functionname)	Im Namens-Server löst den eindeutigen Namen zu einem Socket (IPv4:Port) auf.	Vorbedingung: target ungleich NULL; Rückgabe: String mit Socket-Adresse oder NULL bei Fehler.
void receiveResolution(string servicename, string functionname, string resolvedSocket)		Vorbedingung: servicename ungleich NULL; functionname ungleich NULL ; resolvedSocket ungleich NULL

Tabelle 4.3: Funktionsbeschreibungen Watchdog

Funktion	Beschreibung	Vor-Nachbedingungen
void notifySubscriber-sPeriodically()	interne Funktion, die Subscriber periodisch benachrichtigt	
void checkTimouts()	interne Funktion, überprüft die Gesundheit der bekannten Services	
void heartbeat(string servicename)	Benachrichtigt Watchdog, dass Service noch lebt	Vorbedingung: servicename ungleich NULL;
void subscribe(string servicename, string patternString)	Benachrichtig dem Watchdog, dass der sendene Service über Gesundheit von Services benachricht werden möchte.	Vorbedingung: servicename ungleich NULL, patternString ungleich NULL, ist explizit oder RegEx

4.2 IDL

Die IDL dient der Serialisierung und Zuordnung der RPC Aufrufe. Zur Serialisierung eignen sich

4.2.1 copy in/ copy out

Zum Senden der RPC werden die Nachrichten in eine Nachricht kopiert, damit direkte Speicherzugriffe verhindert werden. Dafür wird ein Format gefunden das für alle Nodes im verteilten System gleich ist. Unter anderem ist JSON ein geeignetes Format. Dieses ist einfach zu verstehen und an für diese Anwendung einen angemessenen Overhead. Zudem ist JSON ein offenener Standard. Andere Formate wie XML haben einen größeren Overhead und sind für die Umsetzung eher nicht geeignet, da auch das Parsen komplexer ist.

Das JSON-RPC ist folgendermaßen aufgebaut:

```
{
 "timeStamp": <int>,
 "function": "<function_name>",
 "params": ["<p1 of Type string>", <p2 of Type int>, "..."]
}
```

Für das Marshalling entsteht somit folgende Tabelle:

Alle Parameter-Typen sind primitive Datentypen und sind ein einem Marshalling-Typ (RpcValue) zusammengefasst. Das ByteArray für die Funktion `updateView` wird aus der Liste von Roboterarmen generiert. Grund dafür ist die Beschränkung, dass die Größe eines einzelnen UDP-Pakets nicht 256 Byte überschreiten darf. Daher gibt es acht 32-bit große Integer, also 256 Bit. Jedes Bit repräsentiert einen Roboterarm. Damit ist R0 Bit 0 von Integer 1, R256 ist Bit 32 von Integer 8.

Funktionsname	**Parameter-Typen**	**Anzahl Parameter**
`resolve`	[string, string, string]	3
`receiveResolution`	[string, string, string]	3
`register`	[string, string, string]	3
`heartbeat`	[string]	1
`subscribe`	[string, string]	2
`reportHealth`	[string, string]	2
`setTimestamp`	[string, string, int]	3
`move`	[int]	1
`select`	[int]	1
`updateView`	[byte[], int, bool, bool]	4
`update`	[byte[], int, bool, bool]	4

Tabelle 4.4: RPC-Funktionen mit Parametertypen

5 Bausteinsicht

In diesem Abschnitt wird die Architektur der Middleware beschrieben.

5.1 Allgemein

Grundlage der Architektur ist eine 3-Layer-Architektur. In dieser ist die Applikation die oberste Schicht, die Middleware die Mittlere Schicht und die OS/Runtime die unterste Schicht. Die Middleware besteht aus einem 3-Tier-Schichten-Modell. Diese ist ein Client-Server-Architektur. Diese Architektur ist eine zustandslose Design, das Skalierbarkeit, Wartbarkeit und Fehlertoleranz dadurch fördert, dass die einzelnen Nachrichten keinen Bezug zueinander haben.

5.2 Beschreibung Client- / Server-Stubs

Jeder Teilnehmer, der die Middleware nutzt hat einen Client-Stub und einen Server-Stub. Als Schnittstelle zur Applikation dient jeweils ein Applikationsstub. Der Applikations-Stub auf Client Seite implemetiert die aufrufenden Funktionen der entfernten Applikation, indem es im Rumpf der entfernten Funktion die Schnittstellen-Funktion „invoke“ aufruft. Diese Funktion ist im Client-Stub implementiert. In der Funktion wird der RPC-Call zunächst durch Marshalling serialisert und mit einem UDP Paket verschickt.

Der Server-Stub empfängt die RPC-Aufrufe über UDP. Das Paket wird im Server-Stub unmarshalled und dann wird die „call“-Funktion aufgerufen. Die „call“-Funktion wird im Applikation-Stub auf der Server Seite implementiert. Dort wird per Callback-Pattern die Funktion der lokalen Applikation aufgerufen. Das wird bei asynchronen RPC-Aufrufen bevorzugt.

5.3 Namensauflösung

Es gibt einen zentralen Namensauflösungsdienst. Dieser implementiert die Funktion „register“ und „resolve“. Der Client des Aufrufers implementiert einen Caching-Proxy, um die Netzwerklast zu reduzieren und um die Verteilungstransparenz zu fördern, da nur der Caching-Proxy den Namensserver kennt. Der Caching-Proxy implementiert die Funktion „receiveResolution“. Der Aufruf von „resolve“ ist synchron. Jeder Eintrag ist für 5 Minuten valide. Damit wird aktzeptiert, dass die Daten für die Auflösung inkonsistent sein kann. Das dient der Echtzeitanforderung und der Safety-Anforderung an die Applikation.

Bausteine

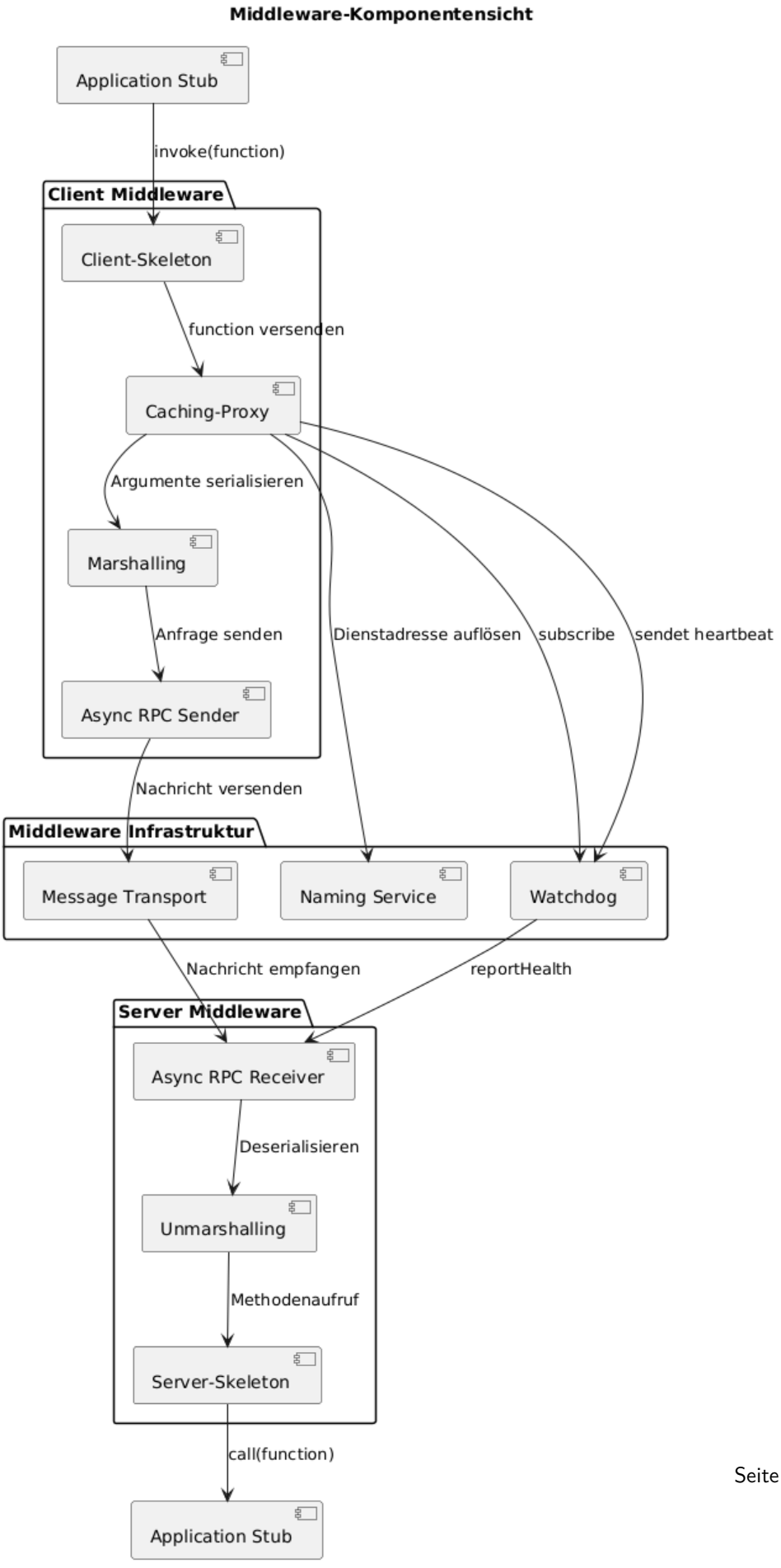

Abbildung 5.1: Komponenten der Middleware

6 Laufzeitsicht

6.1 Szenario U1

Information Weiterleiten

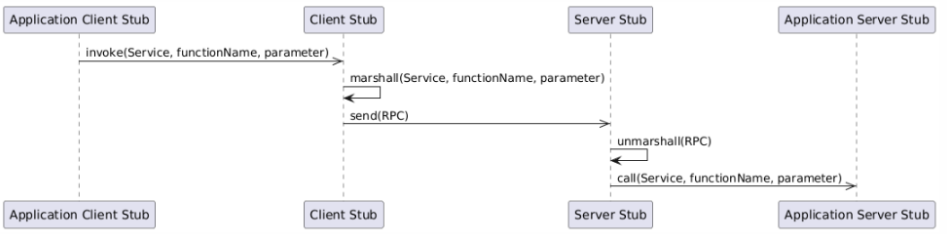

Abbildung 6.1: Information Weiterleiten

6.2 Szenario U2

Node Registrieren

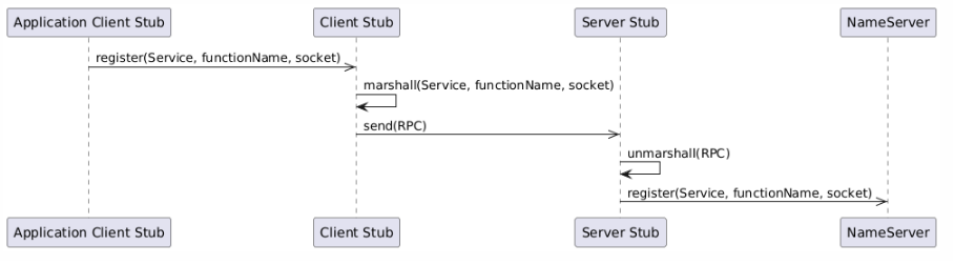

Abbildung 6.2: Node Registrieren

6.3 Szenario U3

Namensauflösung

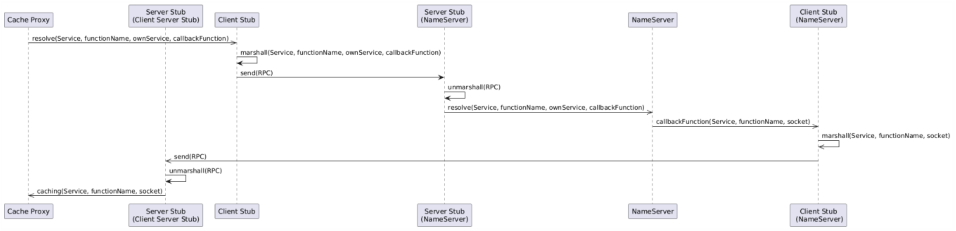

Abbildung 6.3: Namensauflösung

6.4 Szenario U4

Marshalling

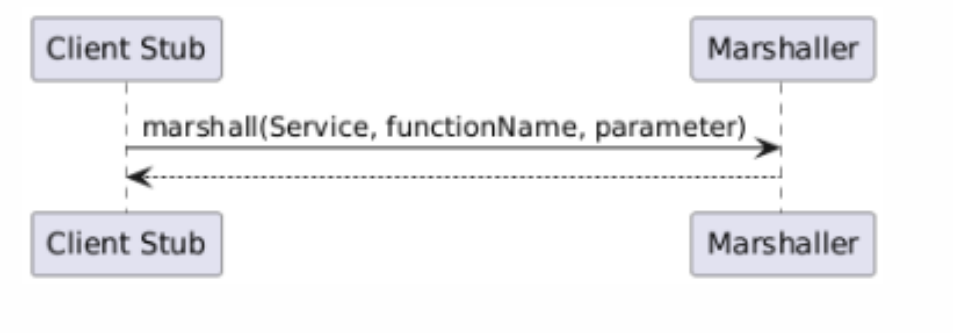

Abbildung 6.4: Marshalling

6.5 Szenario U5

Transport

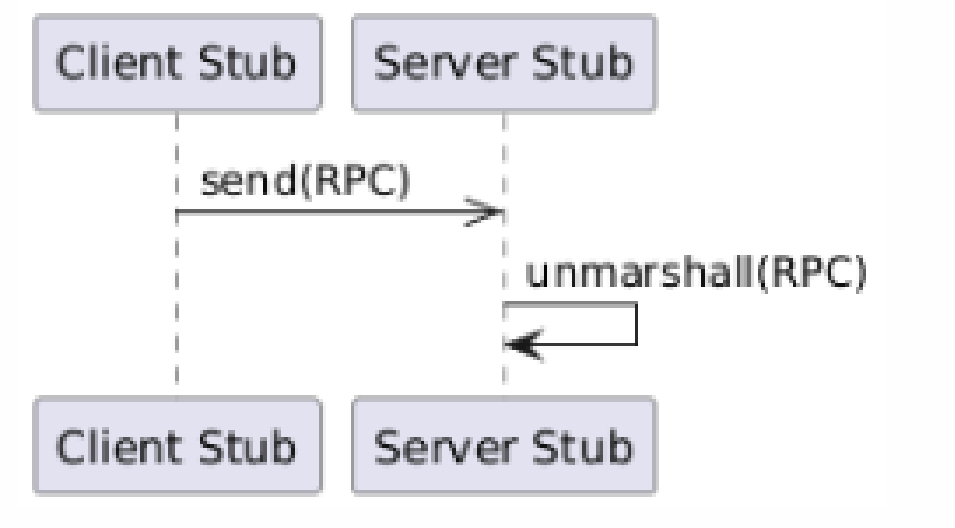

Abbildung 6.5: Transport

6.6 Szenario U6

Sicherheit

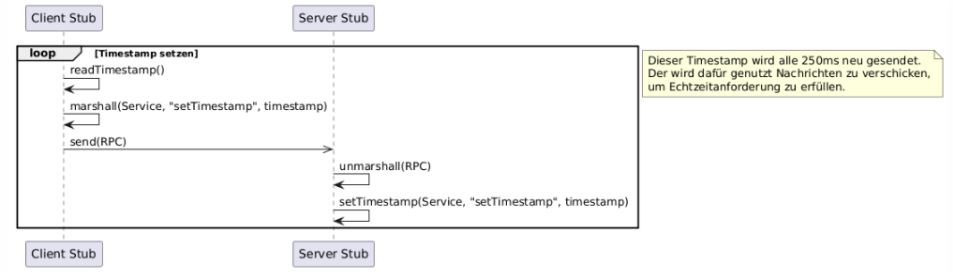

Abbildung 6.6: Sicherheit

6.7 Szenario U7

Watchdog

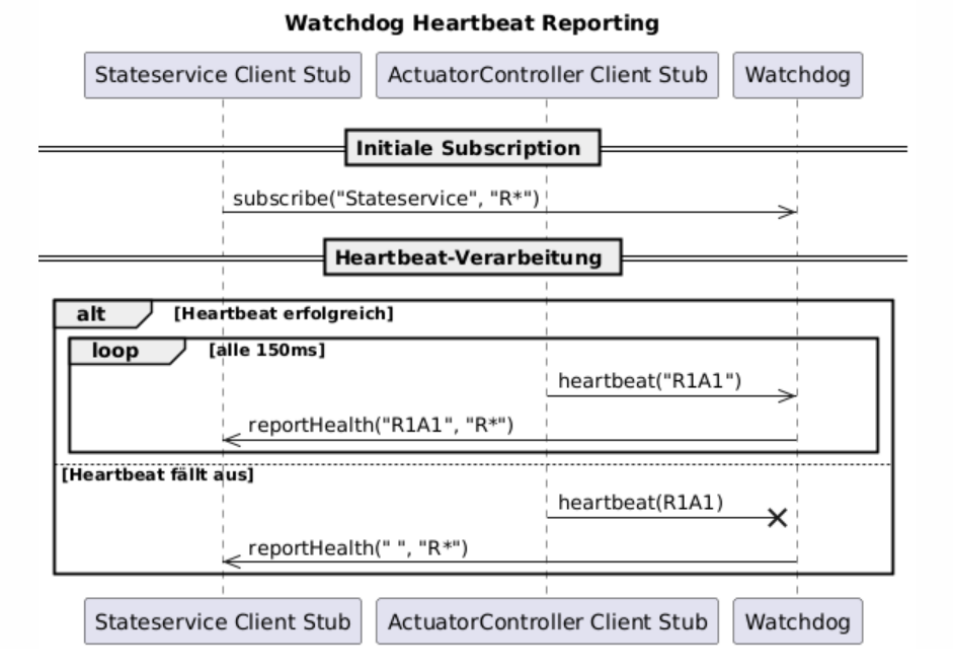

Abbildung 6.7: Watchdog

7 Verteilungssicht

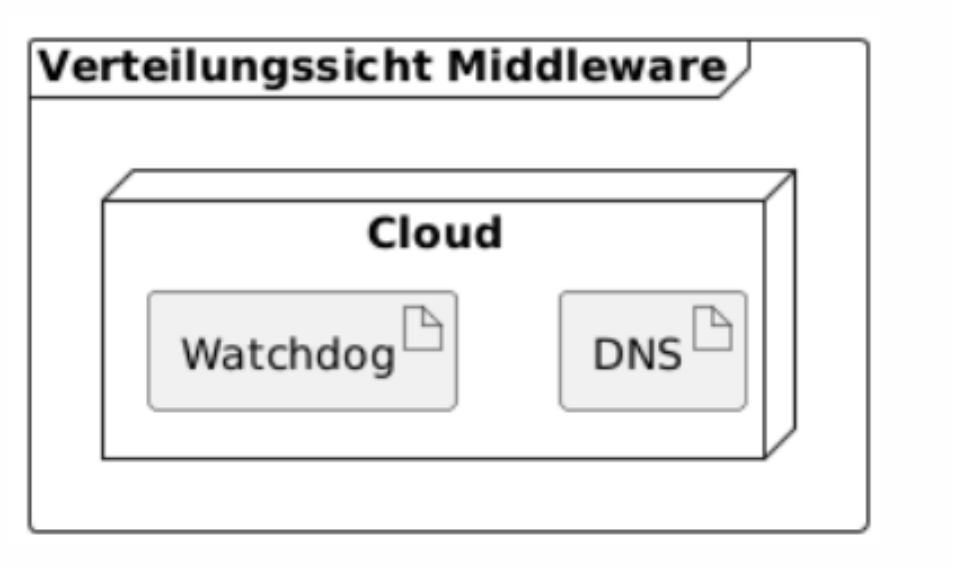

Abbildung 7.1: Watchdog