VSP - arc42 Dokumentation: Middleware

Praktikumsgruppe 02
Ashkan Haghighi Fashi
Minh Lê

31.01.2022 / Version 3.0

Noch unklar wohin damit
Todo:
Kommunikation über TCP und UDP
allgemein RPC Implementierung in Java (als Vorlage)
einfaches Beispiel über RPC

1. Einführung und Ziele

Es wird eine Middleware entworfen nach den Prinzipien die im {Buchtitel} beschrieben werden. Der primär genutzte Dienst ist das Aufrufen von Funktionen auf anderen Nodes.

1.1. Anforderungen (Requirements)

1.2. Qualitätsziele (Quality Goals)

ID	Priority	Quality Goal	Scenario / Erklärung / Beispiele
Q1	1	Scalability	Unser System sollte vorallem skalierbar sein bezüglich der Nutzer/Prozesse
Q2	2	Openness	Die Erreichbarkeit ist hierbei A und O, dafür definieren wir gut Interfaces und sichern Portabilität
Q3	3	Resource Sharing Support	
Q4	4	Distribution Transparency	Jeglicher Zugriff, Ort, Bewegung, Replikation, Ausfälle von Objekten sollte möglichst vor dem Nutzer versteckt werden

1.3. Stakeholder

Role/Name	Contact	Expectations
Entwickler/Architekten/Tester/Betreiber	Minh Lê, Ashkan Haghighi Fashi	Wir halten uns so gut es geht an die Referenzliteratur ● Design Goals ○ Supporting resource sharing ○ Making distribution transparent ○ Being open + scalable
Product Owner	Martin Becke	Ziel: Middleware mit RPC **Anforderungen** ● Openness ○ Anforderungen nach Tanenbaum Foliensack 1 Folie 12 ● Gemeinsame Nutzung von Ressourcen ○ Maximale Auslastung der Rechner von 80% ● Skalierung ○ Geographische: Loopback oder LAN ○ Administrative: 1 Administrator ○ Problemgröße wurde bereits besprochen (2 Spieler => 1 Spiel, N Spiele) ● Verteilungstransparenzen ○ Access ■ Propagieren der Locator/Identifier ○ Relocation ■ Nicht notwendig ○ Concurrent ■ Spiele Sollen sich nicht beeinflussen ○ Failure ■ Kleinere Fehler < 1 sek sind gegenüber dem Kunden zu verschleiern ■ Größere Fehler führen zu einem nicht gewerteten Spiel und zumindest einen Hinweis für den Entwickler

2. Randbedingungen (Architecture Constraints):

Aus den Erwartungen der Stakeholder lassen sich folgende Randbedingungen erschließen:

2.1. Allgemeine Randbedingungen

Randbedingung	Erklärung
Projektmanagement	übliches Vorgehensmodell (Anf., Entw. Impl. Test. Wart.) und verfeinern iterativ, Gitlab

2.2. Technische Randbedingungen

Randbedingung	Erklärung
Design	Sollte erfolgen nach den Prinzipien aus der Referenzliteratur
Repository	HAW-Gitlab

2.3. Organisatorische Randbedingungen

Randbedingung	Erklärung
Abgabetermin	Bis zum 30.01.22 muss alles so gut wie fertig sein
Dokumentation	ARC42 Template
Bearbeitung der Dokumentation	Kollaboratives arbeiten über Google Docs

2.4. Konventionen

Randbedingung	Erklärung
Sprache in den Dokumenten	Englische Klassennamen, englische Methodennamen, deutsche Kommentare, deutsche Dokumentation
Begriffe	Begriffe aus der Referenzliteratur nutzen wie Client Stub, Server Stub, invoke, marshall, etc.

3. Kontextabgrenzung (Context View):

3.1. Business Context:

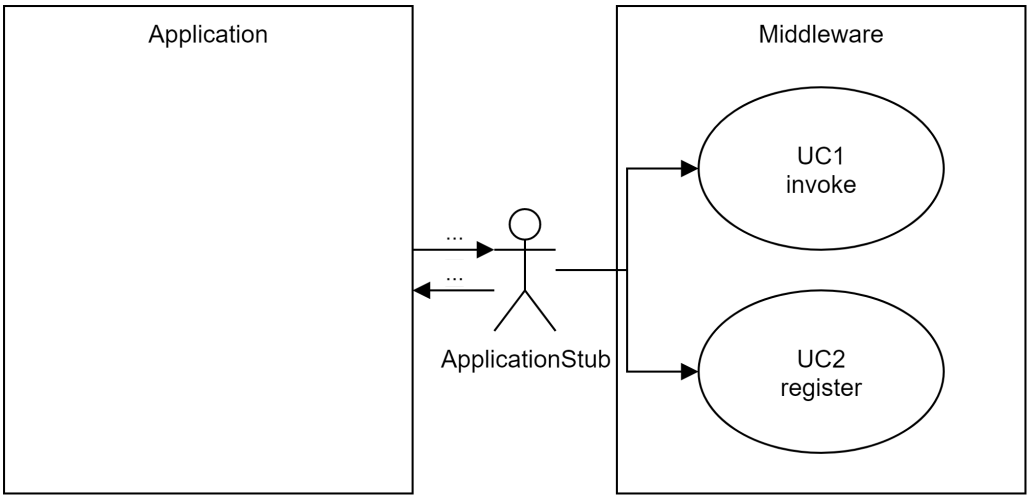

Figure 1. Usecase Diagramm der Middleware

In unserem einfachen Fall wollen Akteure nur die Weiterleitung der Funktionsaufrufe nutzen. Damit Funktionen aufgerufen werden können, müssen sie bekannt sein. Eine Registrierungs zur Laufzeit macht alles dynamischer und einfacher zu benutzen.

Usecase	**Beschreibung**
UC1: invoke	initiiert einen Funktionsaufruf
UC2: register	registriert eine Funktion, damit man weiß, wie man diese Funktion bei wem aufrufen soll

4. Lösungsstrategie (Solution Strategy):

Neben den Funktionen aus den Usecases, wurden Funktionen aufgelistet, die dazu dienen remote Funktionen aufzurufen, und Funktionen zu registrieren.

4.1. Funktionsliste

Akteur	**Methodensignatur**	**Vorbedingung**	**Nachbedingung**	**Semantik**
Consumer	AnyType invoke(MethodCall methodCall)		Dieser Aufruf führt zu einem call(..) Aufruf.	Es wird versucht über die Middleware die Methode mit dem Namen methodCall.method mit den Argumenten methodCall.args aufzurufen, welche ein Teil

				einer Schnittstelle ist, die den Namen methodCall.interfaceName trägt.
Provider	AnyType call(AppStubCallee callee, MethodCall methodCall)	Es wurde invoke aufgerufen.	Die reale Implementierung der Methode wurde lokal aufgerufen.	Es wird durch den callee versucht die Methode auf einem lokalen Objekt aufzurufen.
Provider	register(String interfaceName, AppStubCallee callee)		Der callee ist eingetragen.	Registriert einen Adapter zur Application unter einem Schnittstellennamen.
Provider	register(String interfaceName, Address address)		Ein Name interfaceName kann jetzt zu einer Adresse aufgelöst werden	Registriert eine Schnittstelle in den Name Server
Consumer	Address lookup(String interfaceName)			Führt eine Namensauflösung durch.
Marshaller	Message marshall(MethodCall methodCall)			Packt methodCall in eine verschickbare Nachricht ein
Unmarshaller	MethodCall unmarshall (Message message)			Entpackt einen verpackten MethodCall zu einem MethodCall
	Message serializeReturnValue(AnyType obj)			Verpackt ein Objekt für die Rückgabe in eine Nachricht
	AnyType parseReturnMessage(Message message)			Entpackt eine Nachricht in ein Objekt.

Objektmenge	**Erläuterung**
AnyType	beschreibt einen Typ, der ein Supertyp aller Typen ist, z.B. Object in Java
AppStubCallee	der Adapter, mit dem Funktionen in der Applikation aufgerufen werden können
Address	eine IP-Adresse

Message	Eine Nachricht, die über Sockets versendet werden kann. Siehe nächsten Unterpunkt
MethodCall	fasst String interfaceName, String method, List args zusammen

4.2. Message

Um die Funktionsaufrufe und Rückgabewerte übers Netz zu bringen wird das JSON Format verwendet. Für Argumente und Rückgabewerte haben wir uns überlegt den Objekttypen (oder den primitiven Typen) mit anzugeben. Datentypen wie String, Number, Bool stehen für den jeweiligen JSON primitiv.

Die Serialisierung und Deserialisierung eines primitiven Wertes ist einfach, da alle Java primitive als JSON-Primitiv dargestellt werden können.
Format:

message = {
“type”: primitiver Datentyp als String,
“value”: der Wert als JSON-Primitiv
}

Beispiel: int 6

message = {
“type”: “int”,
“value”: 6
}

Neben primitiven Objekten, haben wir auch ein Format für Listen erstellt.
Wir unterscheiden Listen, deren Elemente einem Typen angehören und Listen wo die Elemente unterschiedliche Typen haben (z.B. eine Parameterliste).

Liste mit einem Typ:

message = {
“type”: Klassenname des Typs,
“values”: [
Jedes Element wird serialisiert, aber nur der message.value Eintrag wird hier übernommen
]
}

Bei der De- / Serialisierung wird nur ein JsonDe- / Serializer verwendet.

Liste mit unterschiedlichen Typen:

message = {
“type”: “?”,
“values”: [
Jedes Element wird serialisiert, und das gesamte message Objekt wird übernommen
]
}

Bei der De- / Serialisierung können mehrere JsonDe- / Serializer verwendet werden.

Ansonsten erlauben wir es eigene JsonSerializer und JsonDeserializer bei der Initialisierung der Middleware zu übergeben.

Auch für MethodCall wurde ein Format festgelegt.

MethodCall:

```
{
        “interfaceName”: obj.interfaceName,
        “method”: obj.method,
        “args”: obj.args
}
```

Bei der Serialisierung wird obj.args wie eine Liste mit mehreren Typen behandelt.

5. Bausteinsicht (Block View):

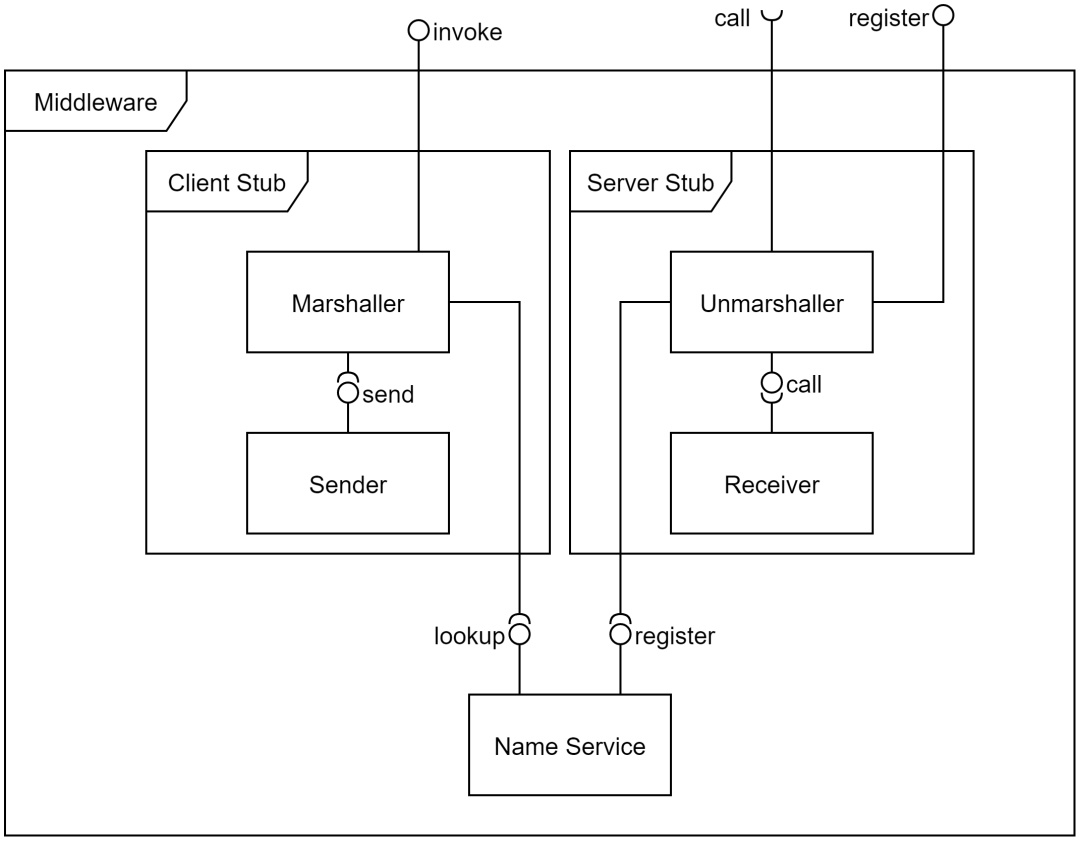

Figure 2. Komponenten der Middleware

Die Komponenten der Middleware, Client- und ServerStub, sind von der Referenzliteratur vorgegeben. Wir haben einen Namensdienst Name Service hinzugefügt, der für die Namensauflösung zuständig ist. Dieses Komponentendiagramm mischt bewusst mehrere Ebenen, denn das Innenleben der Komponenten ist sehr einfach. Aus demselben Grund haben wir Funktionen an die Lollipops geschrieben, anstatt Interfaces. Die meisten Interfaces haben max. 2 Funktionen.

6. Laufzeitsicht (Runtime View):

6.1. invoke() Sequenz

Auf den folgenden Sequenzen, ist dargestellt wie AppStubCaller, Middleware und AppStubCallee einen invoke Aufruf realisieren können.

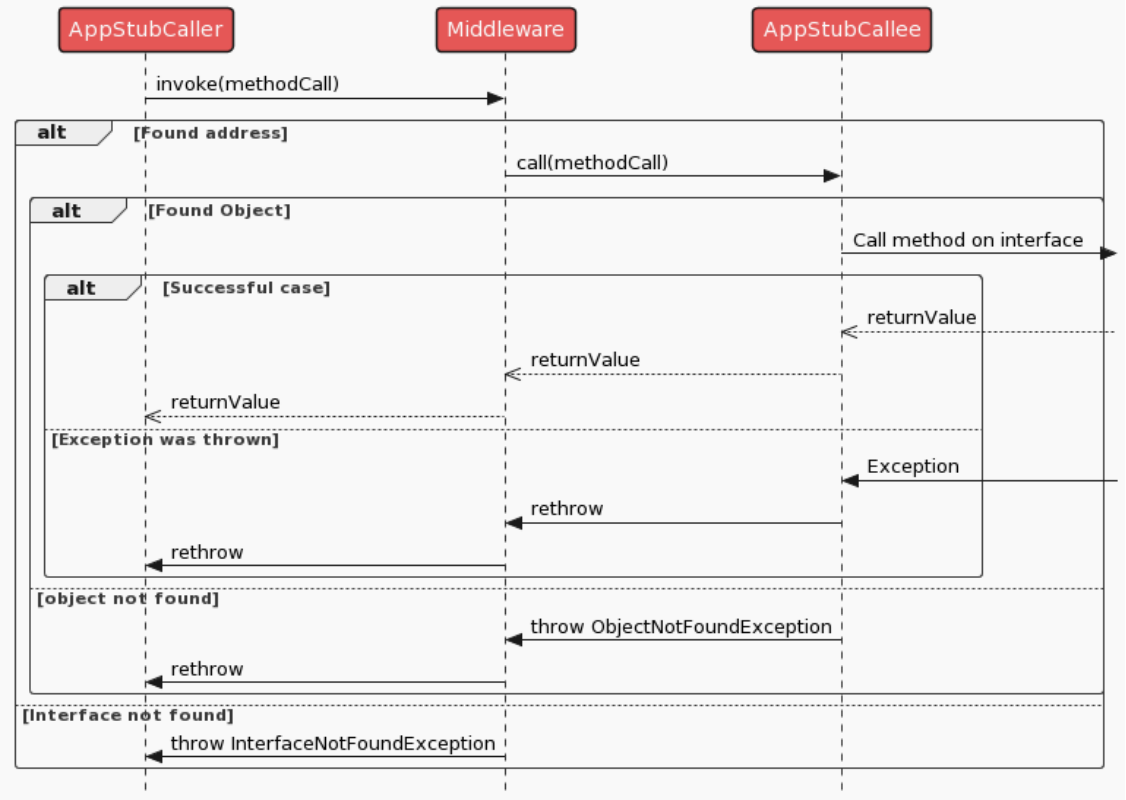

Figure 3. Invoke Sequenz auf oberster Ebene

Ohne die Fehlersemantik, die auf Abb.3 dargestellt wird, besteht ein invoke Aufruf nur aus dem Aufruf an die Middleware, die dann call auf dem AppStubCallee aufruft. Danach wird der Rückgabewert zurückgereicht. AppStubCaller und -Callee sind somit abgedeckt.

Interessanter ist die Sequenz in der Middleware. Auf Abb.4 sieht man wie die 3 Komponenten der Middleware miteinander arbeiten. Der Name Service ist nur für die Namensauflösung da, danach wird das MethodCall Objekt zu einer Nachricht, und an den ServerStub gesendet, den wir mit der Namensauflösung gefunden haben. Dann wird die Funktion lokal aufgerufen und das Ergebnis zurückgesendet.

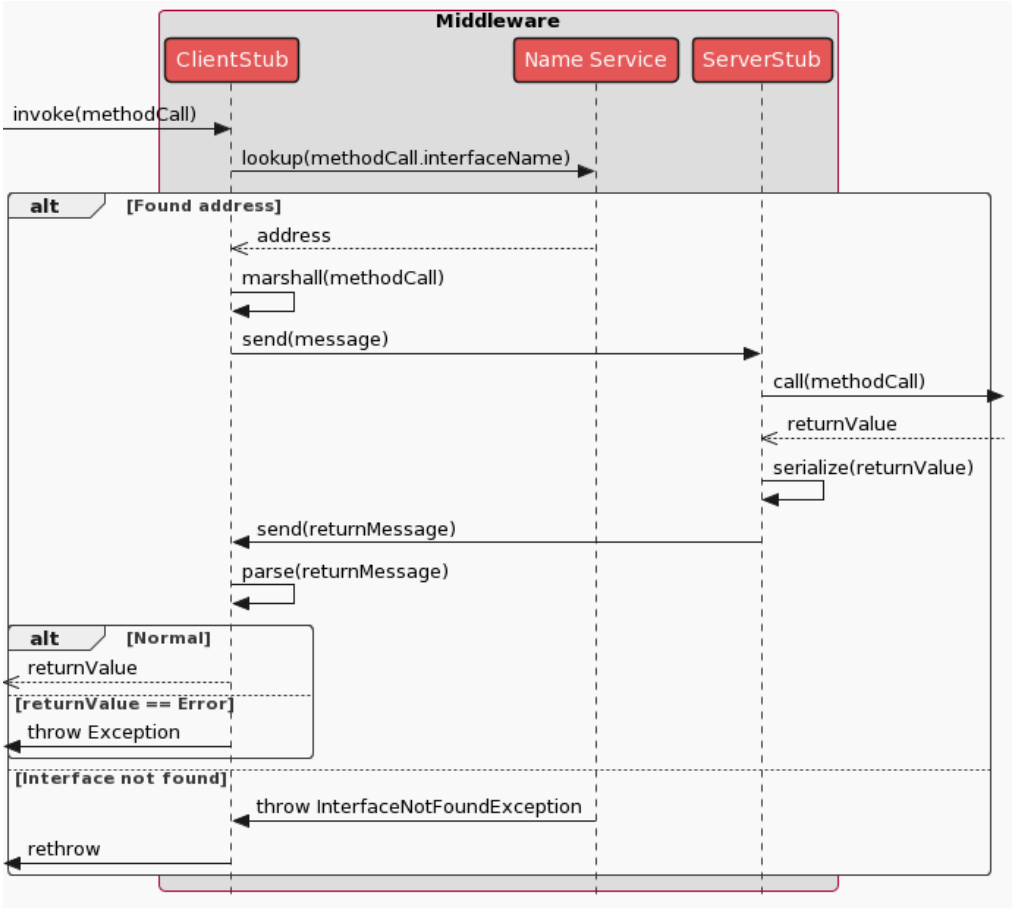

Figure 4. Invoke Sequenz in der Middleware

Das Unmarshalling und der call Aufruf im ServerStub werden auf Abb.5 dargestellt. Nach dem unmarshallen kann überprüft werden, ob das Interface registriert ist. Auf dem zugehörigen AppStubCallee wird dann die Methode ausgeführt.

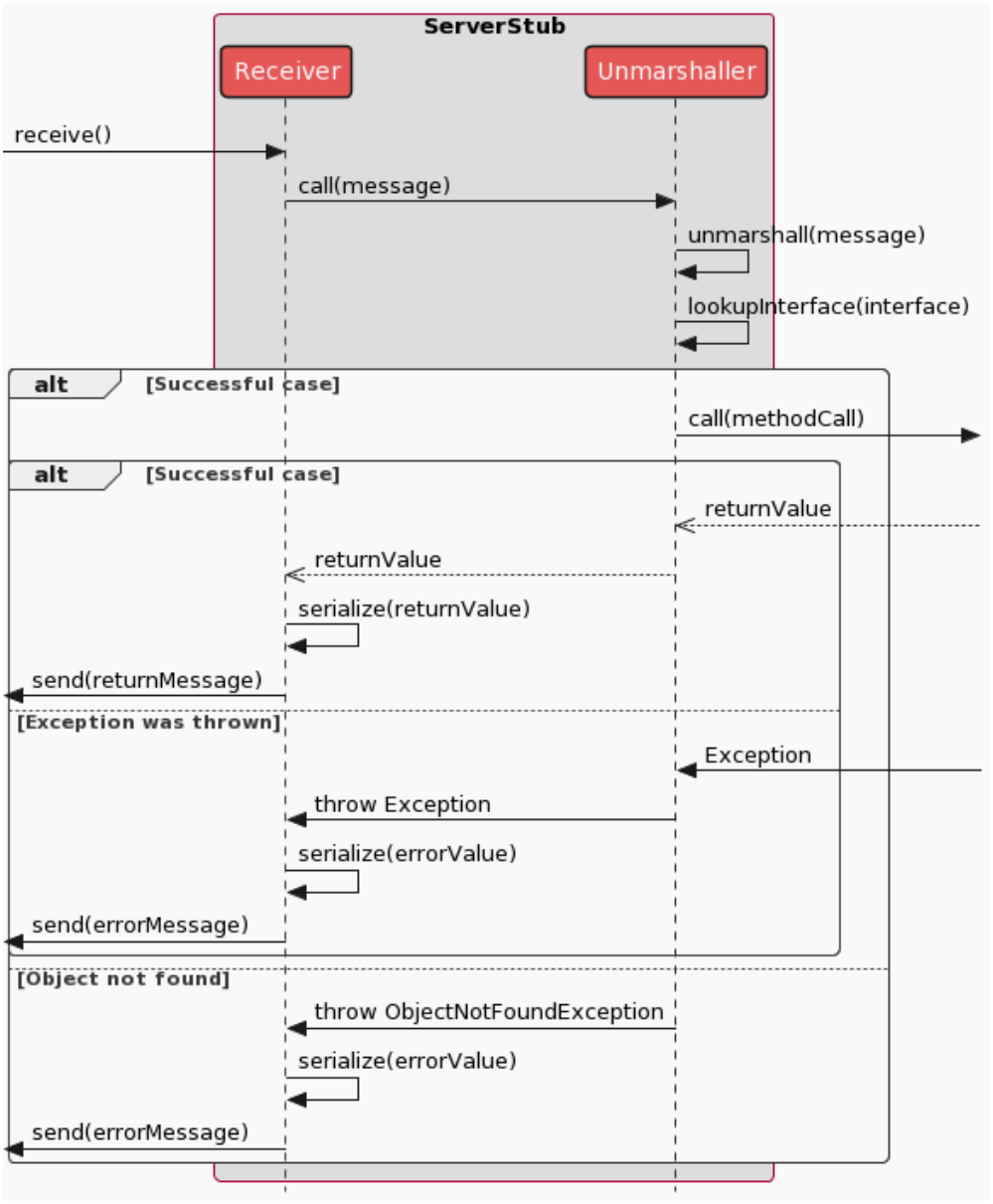

Figure 5. Invoke Sequenz im ServerStub

6.2. call Sequenz

Call ist Teil der invoke Sequenz, und damit in den Sequenzen abgedeckt.

6.3. register Sequenz

Die Sequenz für register wird von einem AppStubCallee Objekt initiiert. In der Middleware ruft der ServerStub den Dienst im Name Service auf. Beides ist in den unteren Abb. dargestellt.

Figure 6. Registrieren einer Schnittstelle aus der obersten Ebene

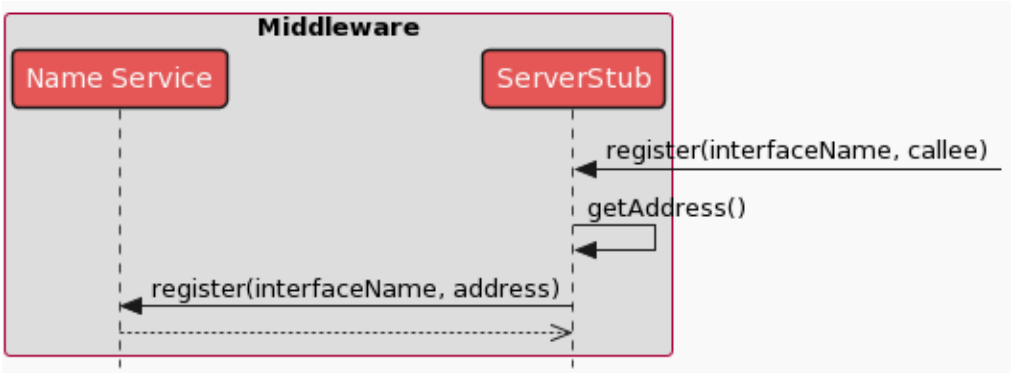

Figure 7. Registrieren einer Schnittstelle in der Middleware

7. Verteilungssicht (Deployment View):

Die Middleware läuft auf jedem Node und es gibt nur einen Name Service Node.

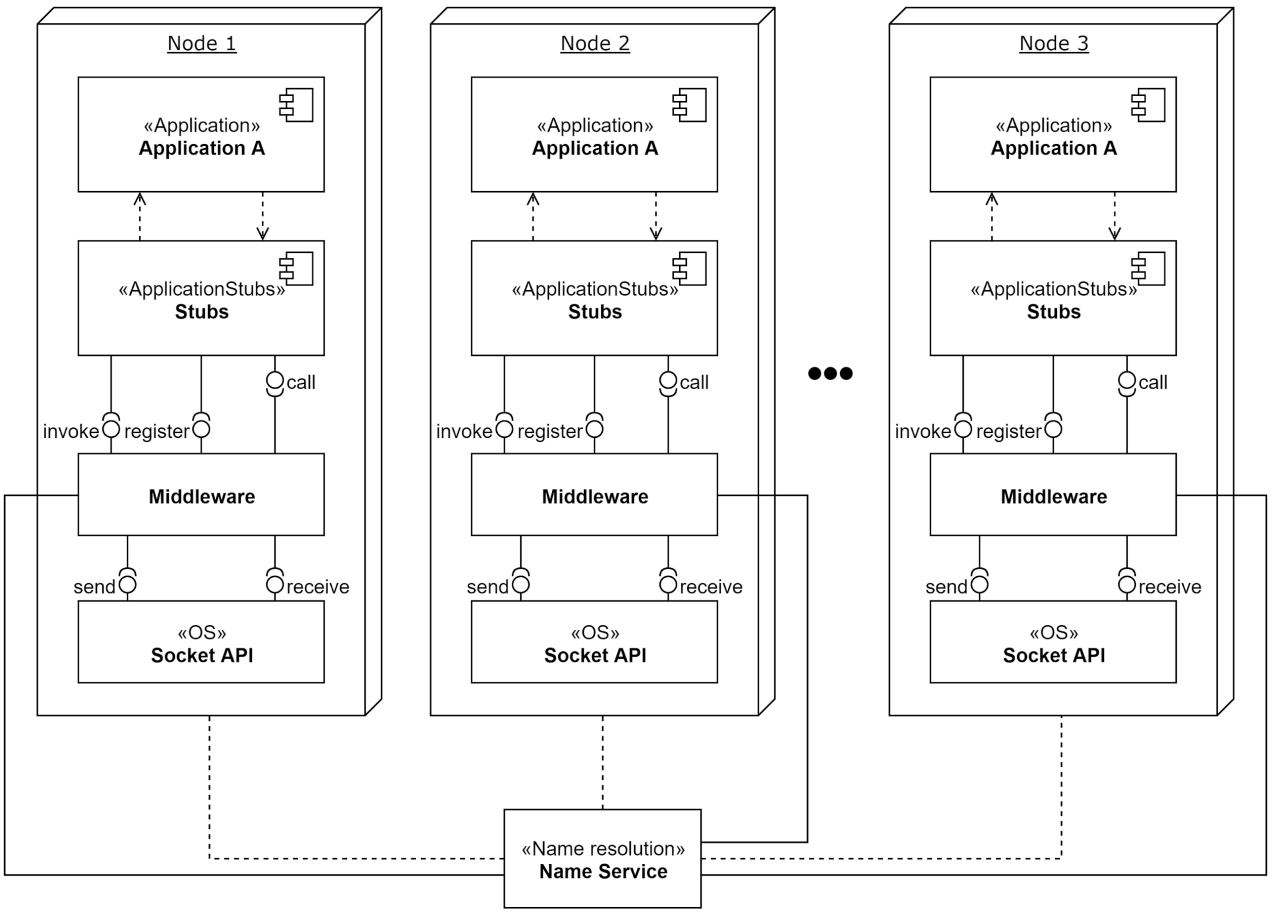

Figure 8. Verteilung in der Middleware