Bachelorarbeit

Till Kahlbrock

Entwicklung eines Dokumentationskonzepts und einer asynchronen Kommunikationsstruktur für ein Fußgänger-Simulationssystem im Rahmen des WALK-Projekts

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Till Kahlbrock

Entwicklung eines Dokumentationskonzepts und einer asynchronen Kommunikationsstruktur für ein Fußgänger-Simulationssystem im Rahmen des WALK-Projekts

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 29. März 2012

Till Kahlbrock

Thema der Arbeit
Entwicklung eines Dokumentationskonzepts und einer asynchronen Kommunikationsstruktur für ein Fußgänger-Simulationssystem im Rahmen des WALK-Projekts

Stichworte
Dokumentationskonzept, arc42, asynchrone Kommunikation, Messaging, RabbitMQ, WALK-Projekt

Kurzzusammenfassung
Diese Arbeit beschreibt die Umsetzung der Kommunikationsstruktur eines Fußgängersimulationsystems. Im Rahmen dieser Entwicklung wird ein Dokumentationskonzept für die Software-Architektur dieses Systems ausgewählt und an die individuellen Anforderungen des Projekts angepasst. Es werden mehrere nachrichtenbasierte Middleware-Produkte untersucht und bewertet.

Till Kahlbrock

Title of the paper
Implementation of a documentation concept and a asynchronous communication structure for a pedestrian simulationsystem in the context of the WALK-Project

Keywords
documentation concept, arc42, asynchronous communication, messaging, RabbitMQ, WALK-Project

Abstract
This thesis describes the implementation of the communication systeme for a pedestrian simulationsystem. In this context a documentation concept for the software-architecture of the simulationsystem is choosen and fitted to the special needs of the project. Different message oriented middleware products are analysed and evaluated in this work.

Inhaltsverzeichnis

1. Einleitung

Die folgenden Anschnitte zeigen die Motivation für diese Arbeit und in welchem Kontext sie durchgeführt wird. Es folgt ein kurzer Überblick über die Ziele der Arbeit und eine Beschreibung der Gliederung.

1.1. Motivation und Kontext

Das WALK Projekt der HAW Hamburg hat sich zum Ziel gesetzt, große Fußgängerströme zu simulieren und die Simulationsergebnisse vielfältig nutzbar zu machen. Die Planung von Großveranstaltungen und Fluchtwegen in Gebäuden, die nachträgliche Analyse von Katastrophen, oder die Live-Unterstützung von Einsatzkräften sind nur einige von vielen Anwendungsgebieten für eine Simulation von Menschenmengen. In WALK werden Menschenmengen nicht wie oft üblich mit Partikelströmen oder durch zellulären Automaten simuliert, sondern mit autonomen Software-Agenten, welche eigenständig und möglichst realistisch handeln (Thiel-Clemen u. a. (2011)). Dieser Ansatz wird als *mikroskopische Simulation* (oder *mikroskopisches Modell*) bezeichnet, da ein sehr detaillierter Blick auf das Verhalten der einzelnen Individuen möglich ist.

Realistisches Verhalten der einzelnen Agenten wird durch individuelle Emotionen, Kognition und einen speziellen sozialen Status erreicht. Details zu einer möglichen Umsetzung dieses Ansatzes liefert Münchow (2012).

Jeder Agent stellt eine eigenständige Softwareeinheit dar, die mit anderen Agenten und ihrer Umwelt kommunizieren muss. In großen Szenarien mit mehreren tausend Agenten führt das zu einem hohen Kommunikationsaufkommen, was wiederum hohe Anforderungen an Performance, Skalierbarkeit, Wartbarkeit und Flexibilität des Systems nach sich zieht. Diese Anforderungen können nur durch eine durchdachte, effektive und sehr gut dokumentierte Software-Architektur erfüllt werden.

Ein bisheriger Ansatz mit einer selbst entwickelten Object Broker Middleware wies einige Defizite bei Flexibilität und Performance auf, sodass eine neue Lösung benötigt wird, die diese Defizite beseitigt und langfristig einsetzbar ist.

1.2. Zielsetzung

Ziel dieser Arbeit ist es, eine Kommunikationsstruktur und ein Dokumentationskonzept zu entwickeln, das das WALK-Simulationssystem in folgender Hinsicht verbessert:

Skalierbarkeit Es sollen Simulationen mit mehreren tausend Agenten durchgeführt werden können. Um dies zu erreichen, sollte das Simulationssystem möglichst gut skalieren und sehr wenig Overhead bei der Kommunikation zwischen den einzelnen Komponenten erzeugen.

Flexibilität Es soll einfach möglich sein, Komponenten auszutauschen oder neue Komponenten dem System hinzuzufügen. Das ist besonders wichtig, da sich das System zur Zeit in einer Aufbauphase befindet und es ständig weiterentwickelt wird.

Performance Die Kommunikation zwischen Komponenten, die auf entfernten Rechnern ausgeführt werden, kann durch verschiedene Faktoren verlangsamt werden. Es ist daher besonders wichtig, dass die Verarbeitung der Nachrichten, die diese Systeme austauschen, mit möglichst geringer Verzögerung durchgeführt werden kann.

Zuverlässigkeit Die Informationen, die zwischen den Komponenten ausgetauscht werden, sind entscheidend für die gesamte Simulation. Werden Informationen verfälscht, oder gehen verloren, kann dies das Scheitern eines gesamten Simulationslaufs bedeuten. Daher ist es sehr wichtig, dass sich der Entwickler einer Komponente darauf verlassen kann, dass die von ihm gesendeten Nachrichten auch ihr Ziel erreichen.

Wartbarkeit Damit Änderungen am System möglichst schnell und mit wenig Aufwand durchgeführt werden können, ist eine gute Architekturdokumentation sehr wichtig. Sie hilft Entwicklern die richtige Stelle für Änderungen zu finden und die Auswirkung der geplanten Änderung abzuschätzen.

All diese Ziele beeinflussen sich gegenseitig und stehen teilweise sogar im Widerspruch zueinander (z.B. Zuverlässigkeit und Performance). Diese Arbeit sucht einen guten Mittelweg, der für das WALK-Projekt den maximalen Nutzen erzeugt und die konkrete Anforderung – die Simulation von vielen Fußgängern – optimal erfüllt.

1.3. Gliederung der Arbeit

Diese Arbeit ist folgendermaßen strukturiert: In Kapitel 2 werden die Grundlagen beschrieben, die für das Verständnis dieser Arbeit wichtig sind. Dabei handelt es sich um eine Definition des Begriffs Software-Architektur, einen Überblick über die Funktionsweise eines Multiagentensystem und die Kommunikation in verteilten Systemen im Allgemeinen und Messaging im Speziellen. In Kapitel 3 wird eine Analyse des Ist-Zustands des WALK-Simulationssystems, der Anforderungen an ein Kommunikationssystem und vorhandener Produkte zur Umsetzung der Kommunikationsinfrastruktur durchgeführt. Des Weiteren werden vorhandene Konzepte zur Dokumentation von Software-Architekturen untersucht und bewertet. In Kaptiel 4 wird das erarbeitete Kommunikationskonzept vorgestellt und auf die Details und Besonderheiten der Lösung eingegangen, in Kapitel 5 wird das Konzept zur Architekturdokumentation vorgestellt. Die Ergebnisse der Arbeit werden in Kapitel 6 evaluiert. Eine Zusammenfassung und ein Ausblick über mögliche Weiterführungen schließen diese Arbeit in Kapitel 7.

2. Grundlagen

In diesem Kapitel werden die Grundlagen zum Verständnis dieser Arbeit vermittelt. Es wird erläutert, was Software-Architektur ist und welche Bedeutung sie für das WALK-Projekt hat. Es folgt eine Beschreibung der Funktionsweise von Softwareagenten und Multiagentensystemen sowie der verschiedenen Möglichkeiten der Kommunikation in verteilten Systemen. Abgeschlossen wird dieses Kapitel mit einer Beschreibung von nachrichtenbasierter Middleware und dem Advanced Message Queuing Protocol (AMQP).

2.1. Software-Architektur

"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them."(Bass u. a., 2003). Dies ist nur eine von vielen Definitionen für Software-Architektur. Das Software Engineering Institute (SEI) der Carnegie-Mellon-Universität versucht auf (Carnegie Mellon Software Engineering Institute) die gängigsten aufzulisten. Die Fülle an verschiedenen Definitionen zeigt, wie schwierig es ist den Begriff exakt zu definieren und wie viele verschiedene Auffassungen von Software-Architektur in der Fachwelt existieren.

Starke (2011) führt einige Punkte auf, die Software-Architektur leisten sollte und auf die sich diese Arbeit größten Teils stützt:

- "Architektur besteht aus Strukturen"
- "Architektur beschreibt eine Lösung" (nicht zwangsläufig den Ist-Zustand)
- "Architektur basiert auf Entwurfsentscheidungen" (und dokumentiert diese)
- "Architektur bildet den Übergang von der Analyse zur Realisierung"
- "Architektur besteht aus verschiedenen Sichten"
- "Architektur schafft Verständlichkeit"
- "Architektur ist der Rahmen für flexible Systeme"

- "Architektur ist Abstraktion"
- "Architektur schafft Qualität"

  Diese neun Punkte geben einen groben Rahmen für die Entwicklung einer guten Software-Architektur und zeigen auf, welche Aufgaben diese erfüllen sollte.

2.2. Multiagentensysteme

Ein *Software-Agent* ist durch sein autonomes Verhalten gekennzeichnet. Er kennt Regeln, um selbständig auf eine sich ändernde Umwelt zu reagieren, und so ein vorgegebenes Ziel zu erreichen. Da es allerdings selten sinnvoll ist nur einen einzelnen Agenten einzusetzen, werden häufig große (manchmal verteilte) Systeme von vielen Agenten verwendet, um bestimmte Aufgaben zu lösen. Man spricht in diesem Zusammenhang von *Multiagentensystemen*.

Wooldrige (2002) bezeichnet Multiagentensysteme als Unterklasse der nebenläufigen Systeme. Viele Agenten arbeiten auf unterschiedliche Weise zusammen: Sie kommunizieren (meist über Nachrichten), koordinieren ihr Verhalten und lösen Konflikte. Sie kooperieren wenn möglich miteinander, um ein gemeinsames oder ihre individuellen Ziele zu erreichen. Dies ist der Hauptunterschied zu klassischen verteilten Systemen, in denen alle Komponenten das gleiche Ziel verfolgen.

Softwaresysteme, die die reale Welt möglichst genau abbilden sind häufig nebenläufig und verrichten viele Aufgaben parallel. In solchen Systemen ist Kommunikation zur Koordination extrem wichtig. Multiagentensysteme entsprechen genau diesen Prinzipien und sind daher ein sinnvoller Ansatz, um Probleme der realen Welt zu simulieren und zu lösen.

Anders als in lokalen Umgebungen können Agenten nicht über Methodenaufrufe kommunizieren. Ein Agent kann einen anderen nicht direkt dazu bringen eine bestimmte Aktion auszuführen, da dieser autonom über sein Handeln entscheidet. Es ist einem Agenten nur indirekt möglich andere Agenten zu beeinflussen. Dies geschieht, indem er eine Nachricht mit einer bestimmten Absicht versendet. Der Empfänger entscheidet dann selbständig, und nach Berücksichtigung seines internen Zustands, über eine Reaktion auf diese Nachricht.

Dieses Verhalten macht es sehr schwer, die Aktionen der Agenten vorauszusehen, das Verhalten des Gesamtsystems wird dadurch aber realistischer und somit aussagekräftiger.

2.3. Kommunikation in verteilten Systemen

Tanenbaum und Van Steen (2007) unterscheidet vier verschiedene Arten der Kommunikation in verteilten Systemen, wobei jede dieser Kommunikationsarten mehrere spezielle Ausprägungen

haben kann. Unterschieden wird in strombasierte Kommunikation, Multicast-Kommunikation, Remote Procedure Call (RPC) und nachrichtenbasierte Kommunikation (*Messaging*). In den folgenden Abschnitten werden diese Kommunikationsarten näher betrachtet und auf ihre Eignung zum Einsatz in WALK untersucht.

2.3.1. Strombasierte Kommunikation

Strombasierte Kommunikation findet hauptsächlich in solchen Systemen Anwendung, die Datenübertragung in einer garantierten Zeit und in einer korrekten Reihenfolge benötigen. Beispiele hierfür sind Audio- und Videoströme, die über das Internet übertragen werden. Hier spielt die Übertragung einer bestimmten Datenmenge in einer zugesicherten Zeit eine große Rolle, da die Qualität des Stroms sonst nicht gewährleistet werden kann. Da WALK solche Anforderungen nicht hat, spielt strombasierte Kommunikation in weiteren Verlauf dieser Arbeit keine Rolle.

2.3.2. Multicast-Kommunikation

Bei Multicast-Kommunikation wird ein Datenpaket von einem Sender zu einer Gruppe von Empfängern gesendet. Diese Art der Kommunikation ist sehr effizient, da der Sender das Paket nur einmal verschicken muss und sich dann die Netzwerkhardware um das Verteilen des Pakets an die Gruppenmitglieder kümmert. Ein ganz entscheidender Nachteil ist, dass das Management der Verbindung über IP-Adressen auf der Internetschicht stattfindet. Es ist also nötig, das Betriebssystem eines jeden beteiligten Rechners auf eine spezielle Weise zu konfigurieren und diese Information auch in der Anwendung parat zu haben. Ein Agent muss dann zum Beispiel wissen, unter welcher Multicast-Adresse er eine bestimmte Gruppe von Agenten erreichen kann.

Dies ist in WALK nur schwer umzusetzen, da Agenten auch während der Simulation zwischen verschiedenen Rechner hin und her wechseln können. Die Gruppe von Agenten die sich zum Beispiel in einem bestimmten Raum befindet ist offensichtlich dynamisch.

2.3.3. Remote Procedure Call (RPC)

RPC ist ein Konzept, dass zuerst in Birrell und Nelson (1984) erwähnt wurde. Die Idee ist es, Prozessen zu erlauben Prozeduren auf entfernten Systemen aufzurufen und diese Prozesse dann solange zu blockieren, bis die Antwort vorliegt. Information können dabei durch Aufruf- und Rückgabeparameter übermittelt werden.

Bei der Implementierung von RPC treten allerdings gewisse Schwierigkeiten auf, die eine Middleware lösen muss. So ist bei einem lokalen Aufruf klar, wo referenzierte Parameter gefunden

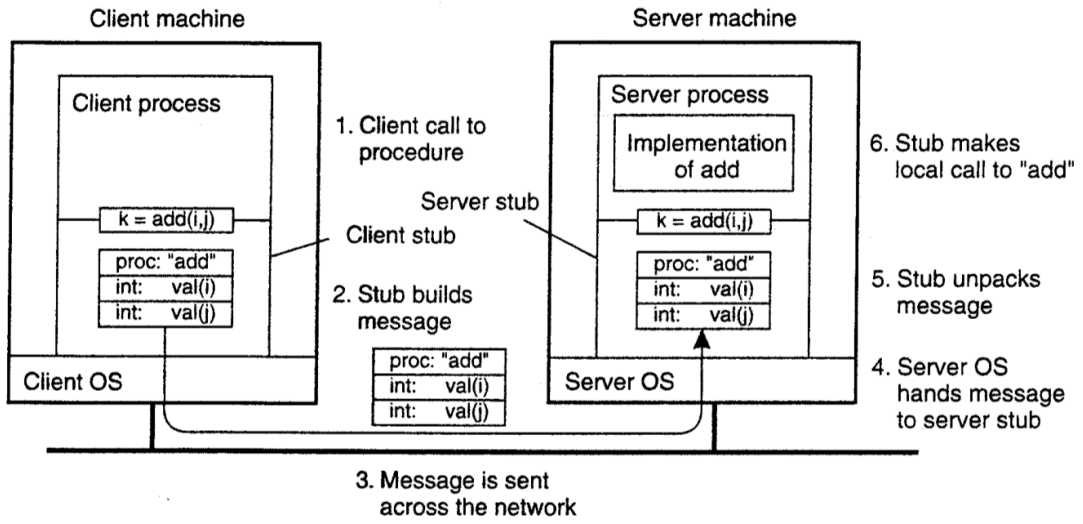

Abbildung 2.1.: Simples Beispiel für einen Remote Procedure Call

werden können. Bei einem entfernten Aufruf kann das aber sehr schwierig sein, da der Aufrufer und die aufgerufene Prozedur in verschiedenen und unabhängigen Adressräumen ablaufen.

Um die entfernten Aufrufe für die Benutzer möglichst wie lokale Aufrufe aussehen zu lassen, werden Client- und Serverstubs eingesetzt. Diese Programme sind die Schnittstellen zu den lokalen Prozessen. Sie implementieren den entfernten Aufruf und bieten dem Benutzer eine saubere lokale Schnittstelle. So wird die Logik zur Kommunikation mit dem entfernten System vor dem Benutzer versteckt und es wird erreicht, dass der Aufruf der entfernten Prozedur wie ein lokaler Prozeduraufruf aussieht.

Abbildung 2.1 zeigt einen beispielhaften Aufruf einer entfernten Prozedur `add(i,j)`, die zwei übergebene Werte addiert.

2.3.4. Nachrichtenbasierte Kommunikation

In manchen Fällen ist die synchrone Kommunikation, die RPCs bieten, hinderlich und kann zu Fehlern oder starkem Performanceverlust führen. Dies ist zum Beispiel dann der Fall, wenn der Sender einer Nachricht nicht weiß, ob der Empfänger gerade online ist und überhaupt zeitnah antworten kann. Ist dies nicht der Fall, wäre der Sender solange blockiert, bis der Empfänger online ist oder ein Zeitlimit überschritten wird.

Um solche Situationen zu verhindern, wurde das Konzept der nachrichtenbasierten Kommunikation (*Messaging*) eingeführt. Die Kommunikation ist dabei typischerweise asynchron. Das heißt, dass ein Sender eine Nachricht sendet und dann nicht – wie bei einem synchronen RPC –

blockiert. Er wartet also nicht auf eine Antwort des Empfängers. Dieses Verhalten führt zwar oftmals zu einem schnelleren Programmablauf, die Implementierung der Kommunikation wird dabei aber auch wesentlich komplexer, da die Ausführung kommunizierender Prozesse nicht mehr so leicht zu synchronisieren ist.

Tanenbaum und Van Steen (2007) unterscheidet zwei Arten der nachrichtenbasierten Kommunikation: 1) flüchtige Kommunikation (*Message-oriented transient Communication*) und 2) persistente Kommunikation (*Message-oriented persistent Communication*).

Ein einfaches Beispiel für flüchtige Kommunikation sind die in allen Betriebssystemen vorhandenen Sockets. Diese bieten eine einfache Programmierschnittstelle zu Netzwerktransportprotokollen (z.B. TCP und UDP). Ein Prozess kann einfach eine Nachricht (also ein Datenpaket) unter Angabe eines Empfängers über das Socket an das Transportprotokoll übergeben. Es wird dann versucht, die Nachricht an den Empfänger auszuliefern. Gelingt dies nicht, wird die Nachricht verworfen und der Sender wird gegebenenfalls (je nach Transportprotokolleigenschaften) über den Misserfolg des Zustellungsversuchs benachrichtigt.

Ein Beispiel für persistente nachrichtenbasierte Kommunikation ist E-Mail. Die Nachrichten werden bei dieser Art der Kommunikation nicht direkt zwischen Sender und Empfänger ausgetauscht, sondern werden vom Sender an eine Message Queue gesendet. Diese nimmt die Nachricht an und stellt sie dem Empfänger zur Abholung bereit. So lassen sich stark entkoppelte und verteilte Systeme realisieren, da sich Sender und Empfänger nicht mehr gegenseitig (wie bei RPC), sondern nur noch die Message Queue kennen müssen. Ein weiterer, sehr wesentlicher Vorteil von persistenter nachrichtenbasierter Kommunikation ist, dass Sender und Empfänger nicht gleichzeitig verfügbar sein müssen. Das macht verteilte Systeme, die auf Messaging anstatt RPC setzten, robuster und verlässlicher.

Abbildung 2.2 zeigt die Architektur eines verteilten Systems, das Nachrichten über Message Queues austauscht.

Abbilung 2.3 zeigt die möglichen Verfügbarkeiten von Sender und Empfänger: a) Sender und Empfänger sind verfügbar. b) Nur der Sender ist verfügbar, die Message Queue speichert die Nachricht, bis der Empfänger sie abholen kann. c) Der Sender hat die Nachricht abgeliefert und ist danach nicht mehr verfügbar. Dies hat keinen Einfluss auf die Zustellung der Nachricht an den Empfänger. d) Die Message Queue speichert die Nachrichten, auch wenn Sender und Empfänger nicht verfügbar sind.

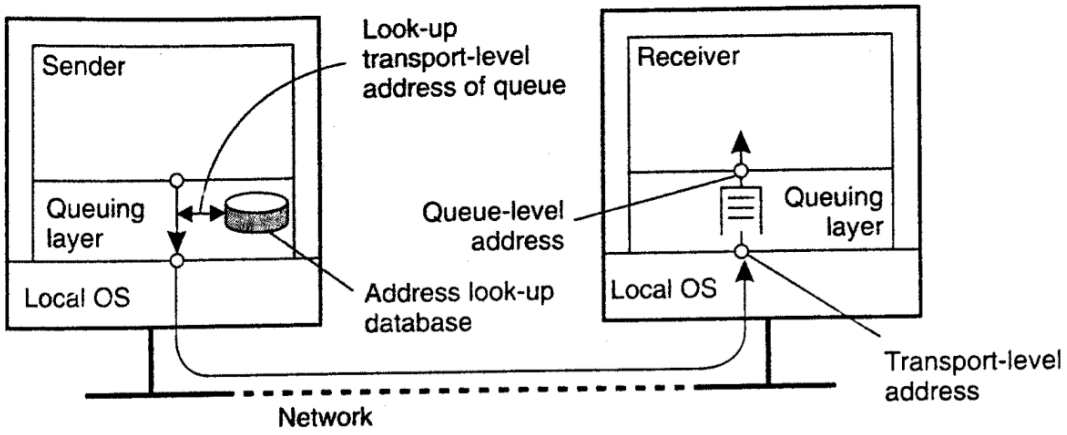

Abbildung 2.2.: Allgemeine Architektur eines verteilten Systems mit Message Queues

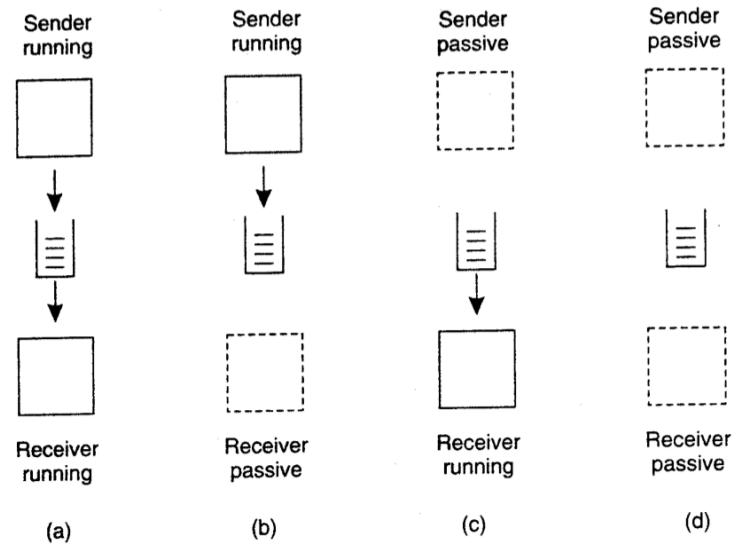

Abbildung 2.3.: Verfügbarkeit von Sender und Empfänger bei nachrichtenbasierter Kommunikation

2.4. Nachrichtenbasierte Middleware

Messaging erlaubt es Applikationen neben ihren Daten, auch ihre Funktionalität zu teilen. Eine Applikation kann die Operationen einer anderen aufrufen, in dem sie dieser eine Nachricht schickt. Der Empfänger führt nach Erhalt der Nachricht eine lokale Operation aus und sendet das Ergebnis gegebenenfalls wieder an den Sender zurück.

Setzt eine Middleware zur Übermittlung von Daten oder zum Aufruf von Funktionalität Nachrichten ein, so spricht man von nachrichtenbasierter Middleware. Nachrichtenbasierte Middleware unterstützt drei verschiedene Kommunikationsarten. 1) *Message passing*, also die direkte Kommunikation zwischen zwei Prozessen über Nachrichten. 2) Beim *Message queueing* werden die Nachrichten an eine Warteschlange gesendet, und dann an einen oder mehrere Empfänger weitergeleitet. 3) *Publish / Subscribe*, also das Senden von Nachrichten an Abonnenten, die sich für eine bestimmte Art von Nachrichten angemeldet haben (vgl. 3.2.1).

Bei jeder dieser Kommunikationsarten ist es möglich einen zentralen Message Broker einzusetzen, oder auf diesen zu verzichten. Die Entscheidung für oder gegen einen Broker hängt meist von der bestimmten Situation ab, da der Einsatz sowohl Vor- als auch Nachteil hat.

Ein *Message Broker* ist der zentrale Punkt in einem brokerbasierten Messagingsystem, an den alle Sender ihre Nachrichten senden. Der Broker stellt die Nachrichten an die Empfänger zu, sodass keine direkte Kommunikation zwischen Sender und Empfänger stattfinden muss.

Der Einsatz von Brokern hat viele Vorteile (vgl. Sustrik (2008)):

- Die Applikationen müssen sich nicht gegenseitig kennen, um miteinander zu kommunizieren. Es reicht, dass jeder den Message Broker kennt und Nachrichten an ihn senden kann. Der Broker leitet die Nachrichten dann anhand von Kriterien wie Queuename, Topic oder bestimmter Nachrichteneigenschaften weiter. Das führt zu einer extrem losen Kopplung.

- Es ist nicht notwendig, dass Sender und Empfänger gleichzeitig online sind. Der Broker speichert die Nachricht des Senders und liefert diese aus, sobald der Empfänger empfangsbereit ist. So werden Kommunikationsfehler durch kurzzeitig ausgefallene oder überlastete Komponenten vermieden.

- Ein Broker kann zur Fehlerresistenz des Gesamtsystems beitragen. Wenn ein sendender Prozess abstürzt, werden seine Nachrichten solange im Broker gehalten, bis er erneut zu senden beginnt. Der Empfänger erhält also die vollständige und korrekte Nachricht und bekommt vom Absturz des Senders nichts mit.

- Der Message Broker kann, wenn nötig, die Nachrichten zwischen verschiedenen Formaten konvertieren. So können unterschiedlichste Applikation verbunden werden, ohne das diese sich vorher auf ein gemeinsames Nachrichtenformat einigen müssen.
- Es ist leicht möglich ein Load-Balancing, Monitoring oder Backup einzurichten, da alle Daten vom Broker an beliebige zusätzliche Ziele weiter geleitet werden können, ohne etwas am Sender ändern zu müssen.

Nachteilig an der Verwendung eines Brokers sind die folgenden zwei Punkte:

- Da jede Nachricht erst an den Broker und dann weiter gesendet wird, entsteht eine wesentlich höhere Netzlast als bei der direkten Kommunikation.
- Ist der Broker überlastet, kann dieser zum Flaschenhals werden und die Leistung des Gesamtsystems negativ beeinflussen. Dieser Nachteil kann durch eine Replikation des Brokers (oft auch als *Clustering* bezeichnet) abgeschwächt werden. Allerdings nimmt dadurch die Komplexität des Systems erheblich zu und es entstehen weitere Kosten durch zusätzliche Hardware und eventuell anfallende Lizenzkosten.

2.5. Advanced Message Queuing Protocol

Das *Advanced Message Queuing Protocol* (AMQP) ist ein offenes Anwendungsschichtprotokoll, das sowohl die Semantik der angebotenen Dienste des Message Brokers definiert, als auch das Format der Nachrichten, die zwischen dem Client und dem Broker ausgetauscht werden müssen. Durch die Spezifikation der Dienste durch das *AMQ model* und des Nachrichtenformats durch das *AMQP*, ist es jeder beliebigen Anwendung möglich, an einen AMQP-basierten Broker Nachrichten zu senden oder Nachrichten von diesem zu empfangen.

2.5.1. AMQ model

Das in AMQP Working Group (2008) beschriebene AMQ model definiert, aus welchen Komponenten ein Broker besteht, wie diese verbunden werden können und wie ein Client diese benutzen kann. Die drei wichtigsten Komponenten sind:

1. Der **Exchange**, der Nachrichten entgegen nimmt und diese an die entsprechende *Message Queue* weiterleitet.
2. Die **Message Queue**, welche Nachrichten speichert, bis ein Client sie abholt, beziehungsweise bis die Nachrichten an den Client ausgeliefert werden können.

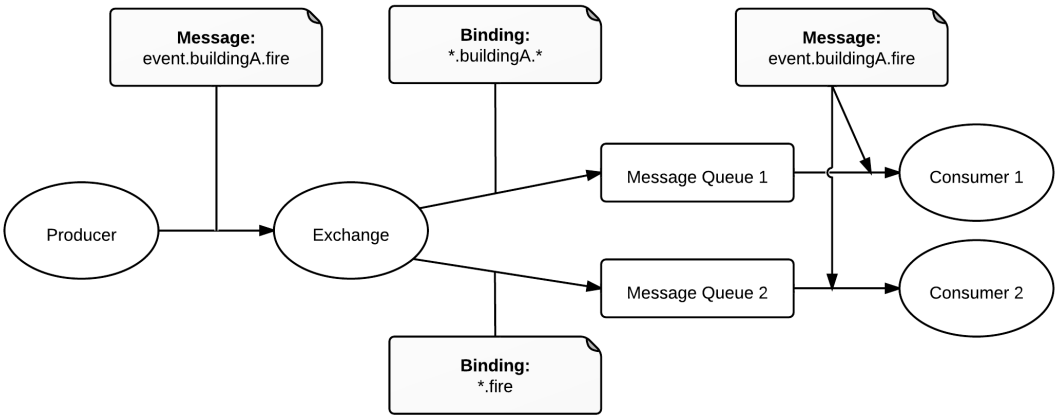

Abbildung 2.4.: Aufbau eines Publish-Subscribe-Systems mit dem AMQP

3. Das **Binding**, das die Verbindung zwischen der Message Queue und dem Exchange herstellt und die Routingkriterien festlegt.

Diese drei Komponenten können auf unterschiedliche Weise kombiniert werden, um die gewünschte Funktionalität zu erzeugen.

Abbildung 2.4 zeigt den Aufbau eines simplen Publish-Subscribe-Systems nach dem AMQP. Es existieren zwei Konsumenten mit je unterschiedlichen Abonnements:

- *Consumer 1* interessiert sich für alle Nachrichten, die Gebäude A betreffen (*Binding: *.buildingA.**).
- *Consumer 2* interessiert sich für alle Nachrichten, die das Ereignis *fire* (*Binding: *.fire*) betreffen.
- Der *Exchange* leitet die Nachricht dann entsprechend der Bindings weiter an die Message Queues der Abonnenten.

2.5.2. AQMP

Das AMQP definiert ein binäres Protokoll, das in zwei Schichten unterteilt ist: 1) Die **funktionale Schicht** definiert Kommandos, welche in verschiedene logische Schichten unterteilt sind (Basic, Transaction, Exchange, Message queues). 2) Die **Transportschicht** ist für den Transport der Daten, das Multiplexing, Framing, Kodieren der Nachrichten, heart-beating und die Fehlerbehandlung zuständig.

AMQP Working Group (2008) gibt als unterstützte Messaging-Architekturen unter anderem *Store-and-forward*, *Publish-Subscribe* und *Punkt-zu-Punkt-Verbindungen* an. Damit ist eine AMQP-basierte Middleware ideal für den Einsatz in WALK, da sich diese Eigenschaften in den in Abschnitt 3.2 definierten Anforderungen wiederfinden.

3. Analyse

Dieses Kapitel untersucht bestimmte Möglichkeiten, die gestellten Aufgaben zu lösen. Dazu wird das WALK-System genauer betrachtet und daraus Anforderungen an ein Kommunikationssystem abgeleitet. Es folgt eine Analyse vorhandener Messaging-Produkte und Dokumentationskonzepte. Ziel dieses Kapitels ist es ein geeignetes Messaging-System und ein Dokumentationskonzept für den Einsatz in WALK auszuwählen.

3.1. Die WALK-Architektur

Die Architektur von WALK ist stark auf die Verteilung der einzelnen Bestandteile des Systems ausgelegt. Dies ist notwendig, da die Berechnung der einzelnen Simulationsschritte sehr viel Rechenzeit benötigt und eine Simulation mit vielen Agenten auf einem einzelnen Computer zu lange dauern würde. Die komponentenorientierte Architektur fördert die Verteilbarkeit und sorgt dafür, dass jede Komponente über sauber definierte Schnittstellen verfügt, über die sie mit anderen Komponenten kommunizieren kann. Abbildung 3.1 zeigt einen groben Überblick über die relevanten Komponenten von WALK und deren Beziehung zueinander.

Im folgenden Abschnitt wird ein kurzer Überblick über die vorhandenen Komponenten und deren Verantwortlichkeiten gegeben.

3.1.1. Geoinformationssystem

Das Geoinformationssystem (GIS) hat die Aufgabe geografische Informationen über Gebäude und Gelände zu speichern und diese für andere Komponenten bereitzustellen. Diese Informationen werden in unabhängigen Schichten verwaltet, um verschiedenste Informationen über eine Lokation zu kombinieren und daraus neue Informationen abzuleiten. Das GIS bietet externen Komponenten eine Schnittstelle, um Informationen über bestimmte Bereiche eines Szenarios (Sektoren) abzufragen. Der Detaillierte Aufbau der GIS-Komponenten wird in Baldowski (2011) beschrieben.

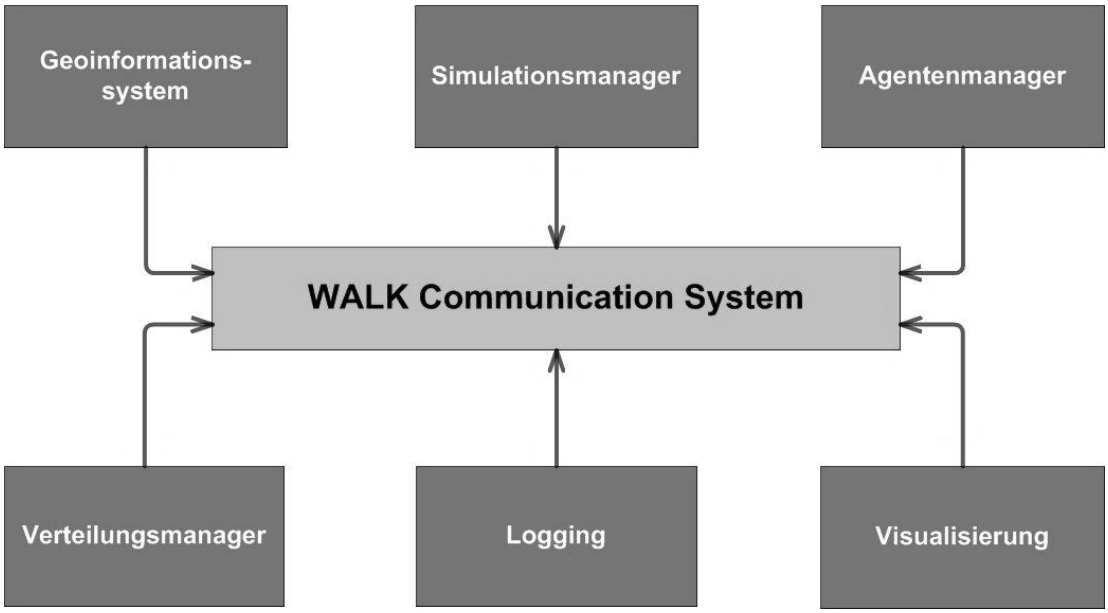

Abbildung 3.1.: Komponenten der WALK-Architektur

3.1.2. Simulationsmanager

Der Simulationsmanger steuert die Simulation und sorgt für einen korrekten Ablauf. Diese Komponente hat eine sehr zentrale Bedeutung, da sich bei ihr alle an der Simulation beteiligten Agenten anmelden müssen. Thiel (2011) nennt als Aufgaben des Simulationsmanagers das Starten und Stoppen von Agenten und Simulationen, das Verändern und Hinzufügen von Szenarien zu einer Simulation sowie die Vorgabe einer globalen Simulationszeit. Diese zentrale Rolle des Simulationsmanagers führt zu einer Serialisierung der Simulation. Diese wird dadurch leichter zu handhaben und Fehler (wie abgestürzte Agenten) können schneller erkannt werden. Der Nachteil ist aber, dass der Simulationsmanager dadurch zu einem potenziellen Single Point of Failure und einem Flaschenhals wird.

3.1.3. Agentenplattform

Die Agentenplattform verwaltet eine Menge von Agenten auf einem physikalischen System. Ihre Aufgaben sind laut Thiel (2011) das Erstellen der verschiedenen Agententypen, sowie das Starten und Beenden der einzelnen Agenten. Sie stellt eine Schnittstelle zwischen Simulationsmanager und dem einzelnen Agenten bereit. Pro physikalischem System existiert genau eine Agentenplattform, die diese Aufgaben übernimmt.

3.1.4. Verteilungsmanager

Der Verteilungsmanager kümmert sich um die sinnvolle Verteilung der Agenten auf die unterschiedlichen physikalischen Systeme. Was in diesem Zusammenhang sinnvoll ist, hängt von den speziellen Anforderungen eines Szenarios ab. Meistens ist aber eine Gruppierung der Agenten nach geografischer Nähe (zum Beispiel alle Agenten in einem Stockwerk) zweckmäßig. So kann die Kommunikation über ein langsames Netzwerk minimiert werden, da sich die Agenten auf einem lokalen System untereinander direkt Nachrichten schicken können.

3.1.5. Loggingmanager

Die Aufgabe des Loggingmanagers ist es, bestimmte Ereignisse zu registrieren und für spätere Analysen zu speichern. Die anderen Komponenten müssen den Loggingmanager nicht kennen, da dieser alle aufgetretenen Ereignisse direkt am Kommunikationssystem mitlesen kann und – je nach Konfiguration – diese dann speichert. Dadurch ist es sehr einfach möglich verschiedene Logger zu benutzen, da diese komplett unabhängig vom restlichen System hinzugefügt oder entfernt werden können. Es könnte beispielsweise ein Logger für Warnungen und einer für kritische Fehler implementiert werden. Die Warnungen könnten einfach in eine Datei geschrieben werden, während die kritischen Fehler eine Mail an eine verantwortliche Person zur Folge haben.

3.1.6. Visualisierung

Eine Visualisierungskomponente hat die Aufgabe die Simulationsergebnisse (z.B. die Bewegung eines Agenten) zu visualisieren. Da die Visualisierung der Ergebnisse auf unterschiedliche Arten geschehen kann, kann es mehrere Visualisierungskomponenten geben. Es sind zum Beispiel 2D- oder 3D-Visualisierungen, oder eine Ausgabe auf einer Webseite denkbar. Es ist dem Entwickler einer Visualisierung somit möglich, eine beliebige Technik einzusetzen, solange diese mit dem Kommunikationssystem kompatibel ist.

3.1.7. Kommunikationssystem

Die Aufgabe des Kommunikationssystems ist der hochperformante und verlässliche Austausch von Informationen zwischen den einzelnen Komponenten. Sie ist die zentrale Komponente der WALK-Architektur und Gegensstand dieser Arbeit. Details zum Kommunikationskonzept und der Realisierung des Kommunikationssystems finden sich im Kapitel 4, eine Beschreibung der Anforderungen an ein Kommunikationssystem in Abschnitt 3.2.

3.2. Anforderungen an ein Kommunikationssystem

WALK besteht aus mehreren Komponenten, die auf verteilten Systemen ausgeführt werden können. Damit der Vorteil der Verteilung der Komponenten – und der daraus resultierenden Erhöhung der Rechenleistung – nicht wieder durch die Verzögerung des Netzwerks aufgehoben wird, ist ein sehr effizientes Kommunikationssystem unerlässlich. Im folgenden werden die Anforderungen, die WALK an ein solches Kommunikationssystem stellt, dargelegt.

3.2.1. Implizite asynchrone Kommunikation

Während einer Simulation treten viele Ereignisse auf, die nur für bestimmte Empfänger von Interesse sind. Um die zu sendende Datenmenge möglichst gering zu halten, ist es wünschenswert auch nur diesen Empfängern die Nachricht über das Auftreten des Ereignisses zu senden. Um ein solches Verhalten zu erreichen, wird das Publish-Subscribe-Muster eingesetzt. Es erlaubt Empfängern, bestimmte Ereignisse beim Kommunikationssystem zu abonnieren (*Subscribe*). Wenn ein abonniertes Ereignis eintritt bekommt der Abonnent eine Nachricht mit bestimmten Informationen, zum Beispiel dem Typ, Zeitpunkt und Ort des Ereignisses.

Möchte ein Sender das Eintreten eines Ereignisses bekannt geben, schickt er eine Nachricht an das Kommunikationssystem (*Publish*). Dieses fasst die Nachricht als Ereignis eines bestimmten Typs auf und benachrichtigt die entsprechenden Abonnenten.

Diese Art der Kommunikation ist sehr effizient, da nur die Empfänger eine bestimmte Nachricht erhalten, die sich auch wirklich für diese interessieren. Es entsteht so einerseits weniger Netzlast, andererseits muss der Empfänger nicht prüfen, ob die erhaltene Nachricht wirklich für ihn ist, da er sie sonst nicht bekommen hätte.

Hinzu kommt, dass eine sehr geringe Kopplung zwischen den Komponenten entsteht. Keine der kommunizierenden Parteien muss die andere kennen. Sie muss nur das Kommunikationssystem kennen, um diesem ihre Publikationen bzw. Abonnements zu senden. Für einen Sender ist es nicht einmal relevant, ob es überhaupt Abonnenten gibt. Gibt es keine, wird das Kommunikationssystem auch keine Nachrichten verschicken.

Abbildung 3.2 zeigt beispielhaft eine solche Kommunikation, bei der ein Rauchmelder in einem Fußballstadion über das Auftreten von Feuer und ein Mensch über das Auftreten von Rauch informiert werden. Hier ist sehr gut die Entkopplung der einzelnen Komponenten zu erkennen, da Mensch (Agent), Rauchmelder (Enviroment / GIS) und Feuer (Ereignis / GIS) miteinander kommunizieren, ohne sich zu kennen. Sie kennen lediglich das Kommunikationssystem und dessen Schnittstellen.

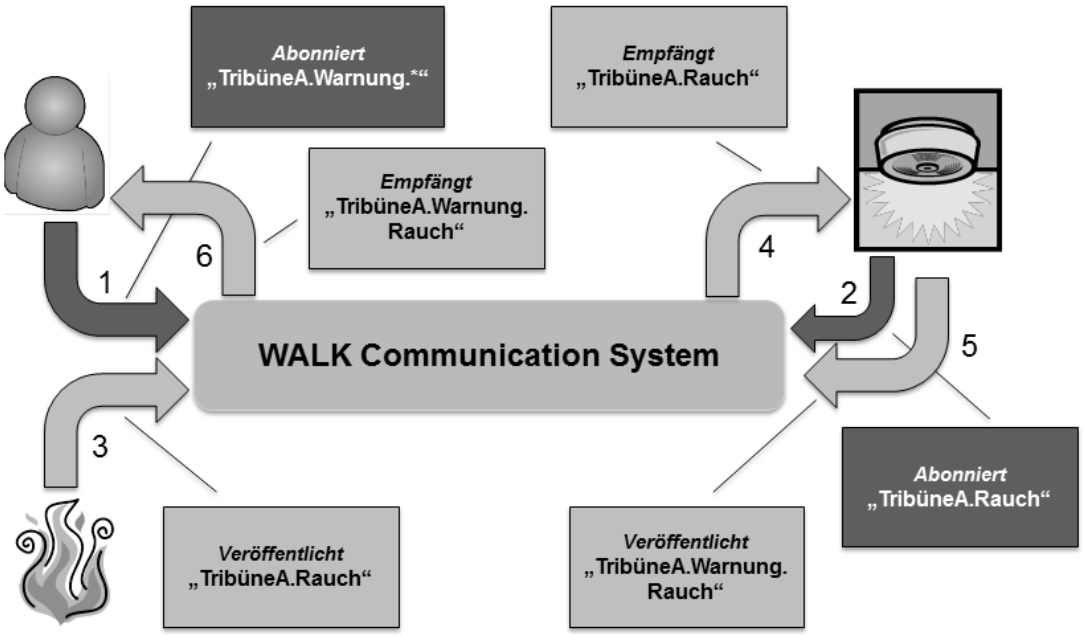

Abbildung 3.2.: Publish-Subscribe-Kommunikation in WALK

3.2.2. Remote Procedure Calls

Remote Procedure Calls sind entfernte Prozeduraufrufe (vgl. Abschnitt 2.3.3). Normalerweise weiß der Aufrufer nicht, dass der Aufruf entfernt ist, da dieser wie ein lokaler Aufruf aussieht und auch so reagiert. Zum Beispiel enthält der Name der aufgerufenen Funktion keine Ortsangaben, oder es werden keine Fehler angezeigt, die auf eine Netzwerkkommunikation hindeuten. Das Wissen, wie der Aufruf vom Aufrufer über das Netzwerk zum Ziel und die Antwort wieder zurück gelangt, liegt bei der Middleware (dem Kommunikationssystem). Tanenbaum und Van Steen (2007) spricht in diesem Fall von Orts- und Zugriffstransparenz.

In WALK wird RPC eingesetzt, um es Komponenten zu gestatten, die Funktionalität einer anderen (möglicherweise entfernten) Komponente zu nutzen. Dies geschieht zum Beispiel, wenn ganz bestimmte Informationen benötigt werden, die von einer anderen, bekannten Komponente bereitgestellt werden.

Abbildung 3.3 zeigt ein Szenario, in dem ein Mensch das GIS nach der Position von Ausgängen in einem bestimmten Raum befragt. Als Antwort erhält er eine Liste der Positionen der Ausgänge des Raums.

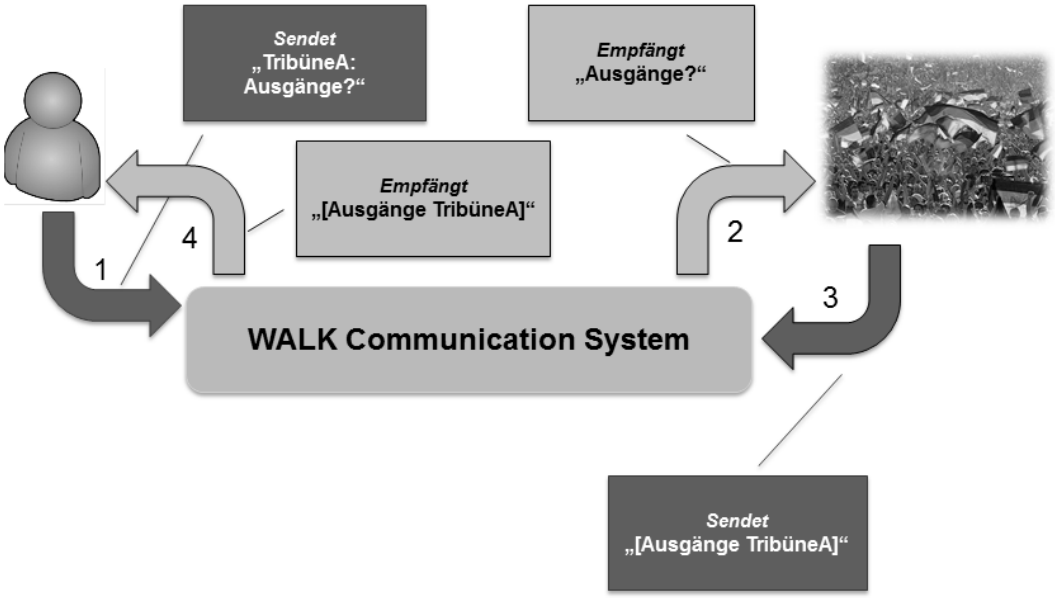

Abbildung 3.3.: Remote Procedure Call in WALK

3.3. Produktübersicht

Es gibt eine Vielzahl nachrichtenbasierter Middleware-Systemen auf dem Markt, die alle mehr oder weniger gut für den Einsatz in WALK geeignet sind. Im folgenden Abschnitt werden drei Systeme betrachtet, die aufgrund verschiedener Eigenschaften besonders gut geeignet sind. Diese Eigenschaften sind vor allem eine liberale Lizenz, gute Performance, eine weite Verbreitung, gute Dokumentation und Interoperabilität. Diese wird zum Beispiel durch die Implementierung des *Advanced Message Queuing Protokolls* (AMQP, siehe 2.5) ermöglicht, das es beliebigen Clients und anderen Messaging-Systemen erlaubt miteinander zu interagieren. Außerdem spielt die Unterstützung von verschiedenen Programmiersprachen eine wichtige Rolle, um nicht durch die Auswahl eines Systems an eine bestimmte Technologie gebunden zu sein. Tabelle 3.1 fast die Eigenschaften der einzelnen Systeme zusammen.

3.3.1. ActiveMQ

AktiveMQ ist ein von der Apache Foundation betreutes Projekt (Apache Software Foundation), das auf dem *Java Messaging Service (JMS)* basiert. Es existieren Clients für alle relevanten Programmiersprachen wie Java, C, C++, C#, Ruby, Perl, Python und viele mehr. ActiveMQ macht auf den ersten Blick einen guten Eindruck, hat aber zwei entscheidende Nachteile: 1)

Tabelle 3.1.: Eigenschaften der untersuchten Message Queues

Eigenschaft	**ActiveMQ 5.5.0**	**ZeroMQ 2.1.11**	**RabbitMQ 2.7.1**
Lizenz	Apache 2.0 License[1]	LGPL[2]	MPL 1.1[3]
Dokumentation	gut	gut	sehr gut
Performance[4]	gut	sehr gut	sehr gut
Anz. offizieller Clients	10	37	11
Persistenz	JDBC	nein	nur eigene DB
AMQP-Unterstützung	nein	Plugin	ja

[1] Apache 2.0 License: http://www.apache.org/licenses/LICENSE-2.0.html
[2] GNU Lesser General Public License: http://www.gnu.org/licenses/lgpl-3.0
[3] Mozilla Public License: http://www.mozilla.org/MPL/1.1
[4] basiert ausschließlich auf eigenen Beobachtungen

Die Performance scheint bei vielen Nachrichten schlechter zu werden (siehe Henjes (2010) und Second Life Wiki (2010)) und 2) bietet ActiveMQ keine Unterstützung für AMQP: “We expect ActiveMQ to implement the latest version, once it is finalized. But, at this time, ActiveMQ does not implement AMQP.” (Apache Software Foundation).

3.3.2. ZeroMQ

ZeroMQ geht einen sehr ungewöhnlichen Weg bei der Realisierung des Messagingsystems. Genau genommen ist ZeroMQ gar keine nachrichtenbasierte Middleware, sondern eine Programmbibliothek, mit der sich nachrichtenbasierte Middleware Systeme leicht implementieren lassen. ZeroMQ setzt dabei auf zwei sehr wichtige Faktoren:

1. ZeroMQ erzeugt viele Prozesse mit nur einem Thread, da so keine langsamen Kontextwechsel nötig sind. “We can move to single-threaded architecture when implementing AMQP. Single threaded processing is dramatically faster when compared to multi-threaded processing, because it involves no context switching and synchronisation/locking. To take advantage of multi-core boxes, we should run one single-threaded instance of AMQP implementation on each processor core. Individual instances are tightly bound to the particular core, thus running with almost no context switches [...].” (Sustrik, 2007).

2. ZeroMQ setzt auf eine Verteilung der Message Queues, sodass diese direkt und ohne zentralen Broker miteinander kommunizieren. Das verhindert, dass der Broker bei vielen Nachrichten zu einem Flaschenhals werden kann.

An diesen Punkten ist zu erkennen, mit welchem Ziel ZeroMQ entworfen wurde: Die Verarbeitung sehr vieler Nachrichten in sehr kurzer Zeit. Mit der von ZeroMQ genutzten Socket-API ist es

möglich, Nachrichten innerhalb eines Prozesses, zwischen unterschiedlichen Prozessen, über TCP oder über Multicast auszutauschen. Dieser sehr flexible und hochperformante Ansatz hat allerdings einen entscheidenden Nachteil: Ein Großteil des Middleware-Codes muss selbst geschrieben und gewartet werden. Das beinhaltet auch sehr wichtige und komplexe Themen wie Transaktionen und garantierte Zustellung von Nachrichten. Bei der Umsetzung eines Projekts mit ZeroMQ muss dieser zusätzliche Aufwand also mit eingeplant werden.

Tests im Rahmen dieser Arbeit haben gezeigt, dass die Entwicklung mit den verfügbaren Client-APIs nicht immer einfach ist. Außerdem befindet sich ZeroMQ in einem beta-Status und besonders der Java-Client macht einen unfertigen Eindruck. Dieser Eindruck wurde durch häufige Abstürze und nicht nachvollziehbare Fehler in einer Testimplementierung gewonnen. Das bedeutet für einen produktiven Einsatz ein sehr hohes Risiko, weshalb ZeroMQ nicht für den Einsatz in WALK in Frage kommt.

3.3.3. RabbitMQ

RabbitMQ ist eine von der Firma Springsource entwickelte Middleware (siehe Springsource), die das AMQ-Protokoll (siehe Abschnitt 2.5) implementiert. Das führt dazu, dass ein RabbitMQ-System extrem interoperabel ist, da durch das AMQP ein beliebiger Client mit diesem kommunizieren kann. Der Server ist in Erlang implementiert und läuft auf der Open Telecom Platform, was zu einer sehr guten Performance durch parallele Verarbeitung vieler Nachrichten zur selben Zeit führt.

RabittMQ bietet Clients für die gängigsten Programmiersprachen wie Java, C#, PHP, Ruby, Python, Erlang und andere. Clients in anderen Sprachen können durch Implementierung des AMQP jederzeit und relativ einfach selbst entwickelt werden.

Die weite Verbreitung, die sehr umfangreiche Dokumentation und die ausgereiften Client-Implementierungen sind gute Argumente für den Einsatz von RabbitMQ. Auch eine sehr gut dokumentierte Plugin-Schnittstelle sowie viele kostenlos verfügbare Plugins (siehe Springsource) sprechen für RabbitMQ. In Kapitel 4 wird der Einsatz von RabbitMQ in WALK im Detail beschrieben.

3.4. Architekturdokumentation

In der Literatur werden zahlreiche Ansätze zur Dokumentation von Software-Architektur erwähnt. Im Folgenden wird ein kurzer Überblick über die wichtigsten Ansätze gegeben, mit dem Ziel, einen Ansatz für die Architekturdokumentation in WALK auszuwählen. Wichtige Kriterien für die Auswahl sind Verständlichkeit, Einfachheit und Flexibilität.

3.4.1. The 4+1 View Model of architecture

1995 beschrieb Phillippe Kruchten in Kruchten (1995) das 4+1-Sichten-Modell zur Dokumentation von Software-Architektur. Das Modell beschreibt die Architektur eines Softwaresystems aus den Perspektiven verschiedener Stakeholder wie Entwickler, Benutzer oder Projektleiter. Nach Clements u. a. (2010) haben diese fünf Sichten die folgenden Funktionen:

- Die **logische Sicht** *(logical view)* beschreibt das System aus Sicht des Endbenutzers und zeigt, wie dieser mit dem System interagiert. Häufig werden für diese Sicht UML-Klassendiagramme, -Kommunikationsdiagramme oder -Sequenzdiagramme benutzt.
- Die **Implementierungssicht** *(implementation view)* beschreibt die Architektur aus der Sicht des Entwicklers und führt die Entscheidungen auf, die für die Implementierung der Software relevant sind. Diese Sicht wird oft mit UML-Paket- oder -Komponentendiagrammen modelliert.
- Die **Prozesssicht** *(process view)* zeigt die dynamischen Aspekte des Systems. Hier werden Prozesse und deren Kommunikation zur Laufzeit dargestellt. Diese Sicht legt besonderen Wert auf Verteilung und Nebenläufigkeit und wird daher am besten mit UML-Aktivitätsdiagrammen modelliert.
- Die **physikalische Sicht** *(deployment view)* zeigt die Anordnung von Softwarekomponenten auf physikalischen Geräten und wird sinnvollerweise mit dem UML-Verteilungsdiagramm dargestellt.
- Die **Anwendungsfallsicht** *(+1 view / use case view)* zeigt Anwendungsfälle und Szenarien, die besonders wichtig für das System und dessen korrekte Funktion sind. In dieser Sicht werden Sequenzen von Operationen zwischen Prozessen beschrieben, anhand derer der Architekturentwurf validiert werden kann. Außerdem kann diese Sicht als Ausgangspunkt für Testfälle oder Architekturprototypen genutzt werden: "The use case view serves as a contract between the customer and the developers and represents an essential input to activities in analysis, design, and test."(Clements u. a., 2010)

3.4.2. TOGAF

The Open Group Architecture Framework (TOGAF) beschreibt einen Prozess, um eine Software-Architektur zu entwickeln und zu dokumentieren. The Open Group (2009) beschreibt TOGAF als Architektur-Framework, das Methoden und Werkzeuge bereitstellt, um Akzeptanz, Produktion, Benutzung und Wartung von Enterprise Architekturen zu unterstützen. Es basiert auf einem

iterativen Prozessmodell und wird durch viele Erfahrungen und Muster zur Entwicklung von Software-Architekturen unterstützt.

TOGAF unterscheidet vier verschiedene Architekturtypen, die als Untermenge einer allgemeinen Architektur angesehen werden können:

1. Die **Geschäftsarchitektur** (*Business Architektur*) definiert die Geschäftstrategie, Unternehmensführung, Organisation und Hauptgeschäftsprozesse.

2. Die **Datenarchitektur** (*Data Architecture*) beschreibt die logischen und physischen Datenbestände und die Datenverwaltung der Organisation.

3. Die **Anwendungsarchitektur** (*Application Architecture*) beschreibt die Interaktionen und Beziehungen der einzelnen Applikationen untereinander und mit dem Kerngeschäftsprozess der Organisation.

4. Die **Technologiearchitektur** (*Technology Architecture*) beschreibt die logischen Software- und Hardwareanforderungen. Dies umfasst auch IT-Infrastruktur, Middleware, Netzwerke, Standards, usw..

Der Prozess zur Architekturentwicklung wird in der *TOGAF Architecture Development Method (ADM)* beschriebem. Die ADM ist ein iterativer Prozess, der aus verschieden Phasen besteht, die nacheinander durchlaufen werden. Die einzelnen Phasen beschreiben das Vorgehen bei der Architekturentwicklung und geben vor, in welcher Reihenfolge die verschiedenen Schritte ausgeführt werden müssen.

> "All of these activities are carried out within an iterative cycle of continuous architecture definition and realization that allows organizations to transform their enterprises in a controlled manner in response to business goals and opportunities." (The Open Group, 2009)

Abbildung 3.4 zeigt die TOGAF ADM. Zu beachten ist hier die *Requirements Management Phase* in der Mitte. Sie bewirkt, dass Änderungen an den Anforderungen (durch den Durchlauf einer früheren Phase) allen späteren Phasen sofort bekannt gemacht werden können.

Starke (2011) beschreibt die Eignung von TOGAF zur Modellierung von Softwaresystemen folgendermaßen: "Meiner Meinung nach eignet sich TOGAF aufgrund seiner größtenteils sehr abstrakten Konzepte kaum zur Modellierung konkreter Softwaresysteme, sondern spielt seine Stärken vielmehr bei der Entwicklung und Modellierung der Unternehmens-IT-Architekturen aus [...]".

Aufgrund dieser Eigenschaften ist TOGAF für den Einsatz in WALK nicht geeignet.

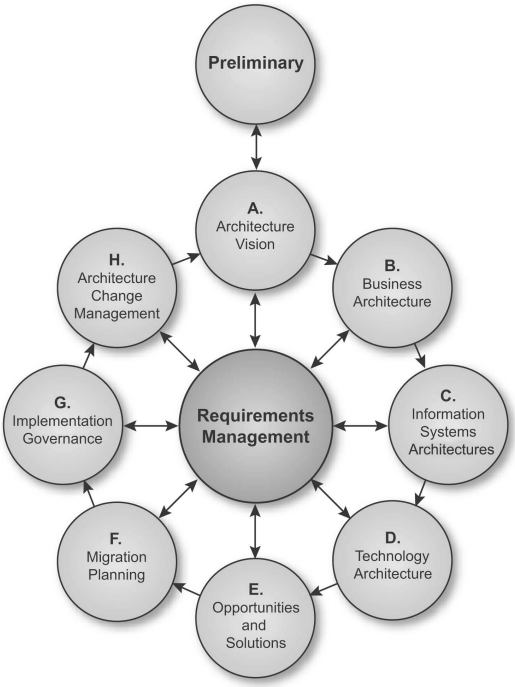

Abbildung 3.4.: TOGAF Architecture Development Method (ADM) Cycle

3.4.3. Das arc42-Template

Das arc42-Template geht weiter als alle bisher betrachteten Architekturdokumentationskonzepte. Es liefert eine Vorlage, die nicht nur die Beschreibung der Architektur durch verschiedene Sichten berücksichtigt, sondern den gesamten Aufbau eines Dokuments vorgibt. Die Vorlage entstammt langjähriger praktischer Erfahrung der Autoren und wird immer wieder überarbeitet und angepasst.

In Starke und Hruschka (2002) beschreiben die Autoren das arc42-Template als praxisnahes Template, das sofort einsatzbereit ist und eine vollständige und strukturierte Sammlung aller architekturrelevanter Themen bildet. Das Template gliedert sich in die folgenden Bereiche:

- Einführung und Ziele der Architektur
- Randbedingung und Einschränkungen
- Kontextabgrenzungen
- Lösungsstrategie und Lösungsansätze
- Die verschiedenen Sichten (Baustein-, Laufzeit- und Verteilungssicht)

- Typische Muster und wiederkehrende Strukturen
- Technische Konzepte
- Entwurfsentscheidungen
- Qualitätsszenarien
- Risiken
- Glossar

Da dieser Ansatz zur Dokumentation von Architektur sehr umfangreich ist und einen sehr guten Rahmen vorgibt, wird er im weiteren Verlauf dieser Arbeit verwendet, um die Architektur von WALK zu beschreiben. Eine detaillierte Beschreibung des Einsatzes dieses Templates in WALK und der vorgenommenen Modifikationen finden sich in Kapitel 5.

3.5. Fazit

Ziel dieses Kapitels war es, sowohl ein geeignetes Produkt zur Kommunikation, als auch ein Konzept zur Dokumentation der Software-Architektur von WALK auszuwählen.

Mit der Auswahl von RabbitMQ als nachrichtenbasierte Middleware und dem arc42-Template als praxiserprobte Vorlage zur Architekturdokumentation ist dieses Ziel erfüllt worden. Die Entscheidung für RabbitMQ ist hauptsächlich durch die weite Verbreitung des Produkts, die sehr umfangreiche Dokumentation und die ausgereiften Client-Implementierungen begründet.

Das arc42-Template ist besonders wegen seines praktischen Ansatzes und des nachvollziehbaren Vorgehens bei der Dokumentation ausgewählt worden.

Die nachfolgenden Kapitel beschreiben wie RabbitMQ und das arc42-Template in WALK zum Einsatz kommen und wie sich der Einsatz von RabbitMQ auf die Performance der Simulation auswirkt.

4. Kommunikationskonzept

In den folgenden Abschnitten wird das Kommunikationskonzept von WALK beschrieben und das *WALK Communication System (WCS)* als beispielhafte Implementierung dieses Konzepts vorgestellt. Das Konzept beantwortet die folgenden Fragestellungen:

1. Wie funktioniert die Kommunikation auf der Server (bzw. Service) -Seite?
2. Wie funktioniert die Kommunikation auf der Client-Seite?
3. Wie ist das Kommunikationssystem intern aufgebaut?
4. Welches Format haben die ausgetauschten Nachrichten?
5. Wie werden Anwendungsfehler erkannt und behandelt?
6. Wie sollten Schnittstellen entworfen werden?

In Abschnitt 4.1 wird eine Service-Implementierung vorgestellt und im Detail beschrieben. In Abschnitt 4.2 werden eine beispielhafte Implementierung eines Clients gezeigt und die wesentlichen Entscheidungen begründet. Abschnitt 4.3 beschreibt die Konfiguration von RabbitMQ sowie den internen Aufbau des Kommunikationsystems. In Abschnitt 4.4 wird der Aufbau der Nachrichten erläutert, die Clients und Services austauschen. Eine Beschreibung der Fehlerbehandlung liefert Abschnitt 4.5. In Abschnitt 4.6 folgt dann eine kurze Erläuterung zum Entwurf von Schnittstellen.

4.1. Service

Einige Komponenten in WALK bieten Dienste für andere Komponenten an. So ist es beispielsweise möglich, Informationen über bestimmte geografische Gebiete der simulierten Welt bei der GIS-Komponente abzufragen. Der folgende Methodenaufruf liefert den Hauptsektor einer Simulation, von dem aus alle anderen Sektoren über eine Baumstruktur erreicht werden können:

```
Sector sector = gis.getRootSector(simulationId);
```

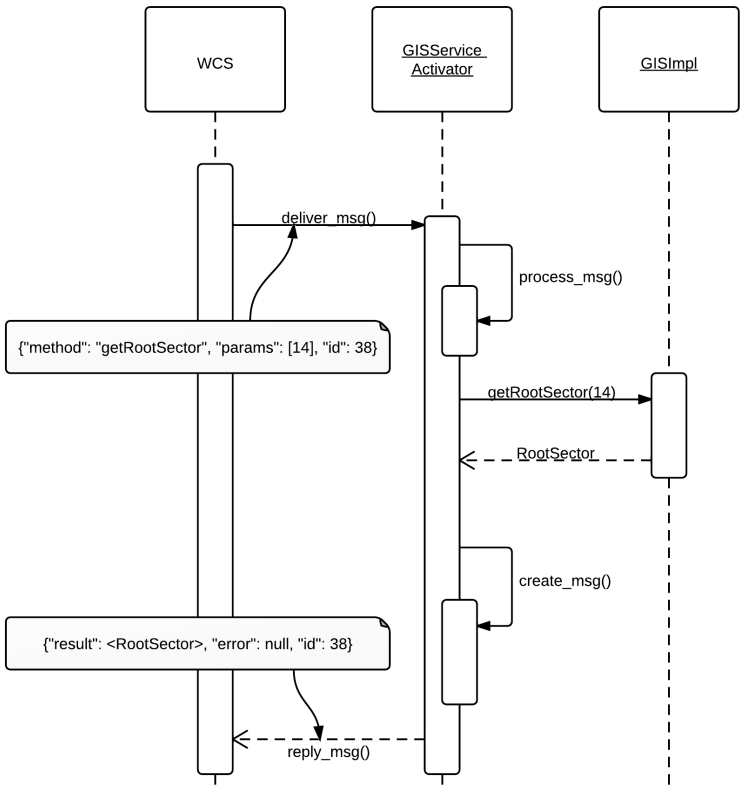

Abbildung 4.1.: Arbeitsweise des Service-Activator-Patterns am Beispiel eines GIS-Aufrufs

Da diese Information für die verschiedensten Komponenten von Interesse sein kann, ist es nicht möglich vorauszusagen, wer diese Operation aufrufen wird. Es wird also eine Schnittstelle benötigt, die von einem beliebigen Client genutzt werden kann. Dabei ist zu beachten, dass alle Komponenten beliebig verteilbar sind. Es muss also möglich sein, diese Schnittstelle sowohl lokal als auch entfernt zu benutzen. Außerdem soll der Aufrufer entscheiden, ob der Aufruf synchron oder asynchron erfolgen soll.

Um diese Anforderungen zu erfüllen, wird das in Hohpe und Woolf (2004) beschriebene *Service-Activator-Pattern* eingesetzt. Der Service-Activator nimmt die Anfrage über das Messaging-System entgegen und ruft die eigentliche Komponente lokal auf. Falls eine Antwort erforderlich ist, wird diese wieder in eine Nachricht verpackt und an den Absender zurückgeschickt. Abbildung 4.1 zeigt diesen Ablauf am Beispiel des oben genannten Aufrufs des GIS.

Der Service-Activator agiert als zusätzliche Schicht zwischen entfernten Aufrufen und der lokalen Implementierung des GIS. Lokale Aufrufe an das GIS können ganz normal durchgeführt

werden, ohne den Serive-Activator zu benutzen. Das führt zu einer sauberen Trennung von technischen Messaging-Code im Service und fachlichem Code im GIS.

4.1.1. MQClient

Der RabbitMQ-spezifische Code wird im *MQClient* versteckt. Dieser bietet den Teilen des Systems, die mit dem Messaging-System kommunizieren müssen, eine allgemeinere Schnittstelle an, sodass diese keine RabbitMQ-Bibliotheken einbinden müssen.

Die Schnittstelle des MQClient umfasst die folgenden Operationen:

- `void publish(String topic, String message)`: Sendet eine Nachricht an ein bestimmtes Topic
- `void subscribe(String topic, Callback callback)`: Registriert einen Callback für ein bestimmtes Topic
- `void reply(String response, String corrId, String replyTo)`: Sendet eine Antwort mit einer bestimmten *correlation id* an den Absender einer Anfrage
- `<T> T call(String basicRoutingKey, String methodName, List<Object> params)`: Macht einen synchronen Methodenaufruf und liefert die Antwort als Typ T

Die Operationen sind möglichst allgemein gehalten. Damit ist es ohne Weiteres möglich, das zugrunde liegende Messaging-System durch ein anderes zu ersetzen ohne den Applikationscode zu ändern. Abbildung 4.2 zeigt die Integration des MQClient in das System. Eine beispielhafte Implementierung des MQClients in Java zeigt Anhang B.

4.1.2. Nameservice

Die Verwendung eines Nameservice ist in WALK nicht notwendig, da alle Services über die zur Verfügung gestellten Clients angesprochen werden können (s. Abschnitt 4.2). Es ist lediglich notwendig, jedem System mitzuteilen, unter welcher Adresse das Kommunikationssystem zu erreichen ist. Dies kann zum Beispiel über Konfigurationsdateien oder Startparameter geschehen und hängt von der verwendeten Technik zur Realisierung der einzelnen Komponenten ab.

4.2. Client-Library

Eine Client-Library ist ein *Application Programming Interface (API)* um Client-Anwendungen für bestimmte Services zu erstellen. Im Normalfall wird die Client-Library vom Autor des Services

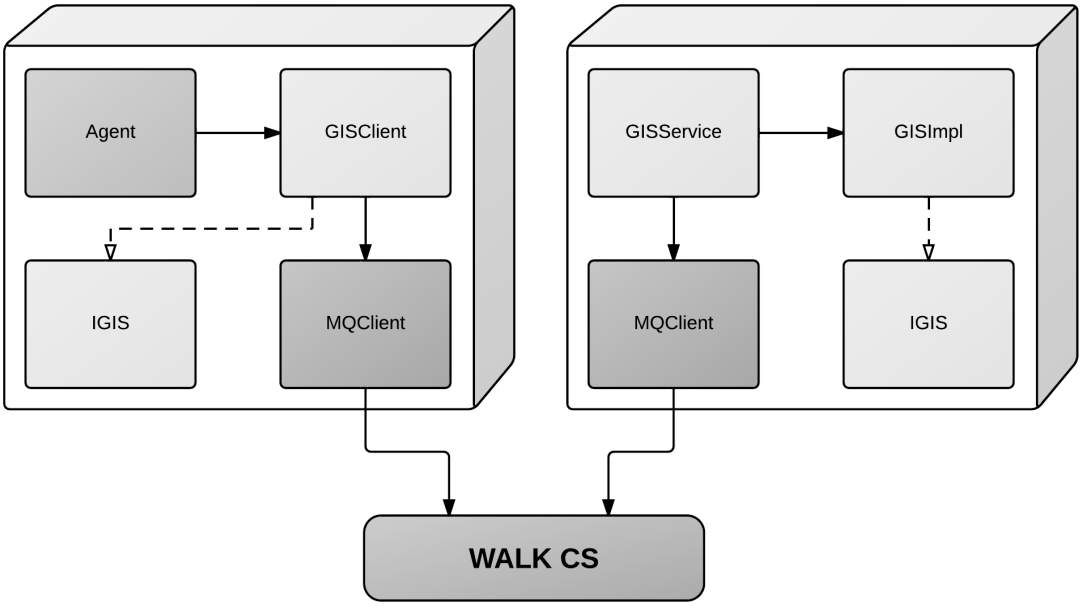

Abbildung 4.2.: Integration des MQClient in das WALK-System

erstellt, sodass ein Programmierer keine genaue Kenntnis der Funktion des Services benötigt, um Client-Anwendungen zu erstellen.

Ziel der Verwendung einer Client-Library ist es, dem Nutzer eines Services eine möglichst einfache Schnittstelle zur Verfügung zu stellen und die Kontrolle über die technischen Details vor ihm zu verbergen. So hat der Nutzer eine stabile Schnittstelle und kann sich auf die Entwicklung des Clients konzentrieren.

Eine Alternative zu einer Client-Library ist die Verwendung einer *Interface Definition Language (IDL)* wie zum Beispiel der *Web Service Description Language (WSDL)*. Sie beschreibt die Schnittstelle eines Services sehr allgemein mit XML und ist so für beliebige Anwendungen zu benutzen. Allerdings liegt hier die Verantwortung zur korrekten Umsetzung der technischen Details (wie zum Beispiel Nachrichtenformaten und Protokollen) beim Nutzer des Services. Außerdem steht die so gewonnene Flexibilität einer schlechteren Performance gegenüber, da die Schnittstellenbeschreibung von jedem Client erneut interpretiert werden muss.

Abbildung 4.2 zeigt, wie ein Client (*Agent*) eine Client-Library (*GISClient*) verwendet. Eine beispielhafte Implementierung einer Client-Library und eines Client finden sich in Anhang A.

4.3. Das WALK Communication System

Das *WALK Communication System (WCS)* basiert auf drei wesentlichen Teilen:

1. Den Services in den jeweiligen Komponenten (siehe Abschnitt 4.1),
2. den Client Libraries (siehe Abschnitt 4.2)
3. und dem RabbitMQ-Server

Der RabbitMQ-Server fungiert als Message Broker und stellt den Komponenten von WALK einen zuverlässigen und schnellen Kommunikationskanal zur Verfügung. RabbitMQ implementiert das AMQP 2.5 und setzt dessen strukturelle Vorgaben um. Es existieren Exchanges, Queues und Bindings.

In WALK werden die beiden unterschiedlichen Arten der Kommunikation (RPC und Pub/Sub, siehe Abschnitt 3.2) mit einem Exchange und vielen Queues realisiert. Es gelten die folgenden Konventionen:

- Es wird nur der Exchange *wcs* benutzt
- Der Exchange *wcs* ist vom Typ *topic*
- Bindings für RPC haben das Schema: *rpc.<komponentenname>.<methodenname>* (z.B.: rpc.gis.getRootSector)
- Bindings für Events (Pub/Sub) haben das Schema: *event.<typ>.<area1>.<area2>.....<arean>* (z.B.: event.fire.bt7.hausA.r311)
- Alle RPCs werden als JSON-RPC versandt (siehe Abschnitt 4.4)

Abbildung 4.3 zeigt die Struktur von Exchange, Queues und Bindings des WCS am Beispiel des GIS. Es gibt einen Producer (*Agent*) und zwei Consumer (*GIS* und *RPCLogger*). Jeder Consumer hat seine eigene Queue, die durch spezielle Bindings mit dem Exchange verbunden wird. Das GIS interessiert sich nur für Nachrichten, die mit dem Routingkey *rpc.gis.* beginnen, wohingegen der RPCLogger alle RPCs empfangen möchte (*rpc.#*). Antworten des GIS auf RPCs werden über den default exchange an die temporäre Queue *XYZ* gesendet.

Durch diese Struktur ist es ohne Schwierigkeiten möglich, neue Komponenten hinzuzufügen und diese Nachrichten empfangen oder senden zu lassen. Durch eindeutige Konventionen ist sichergestellt, dass es keine Konflikte beim Senden und Empfangen von Nachrichten gibt. Sollte

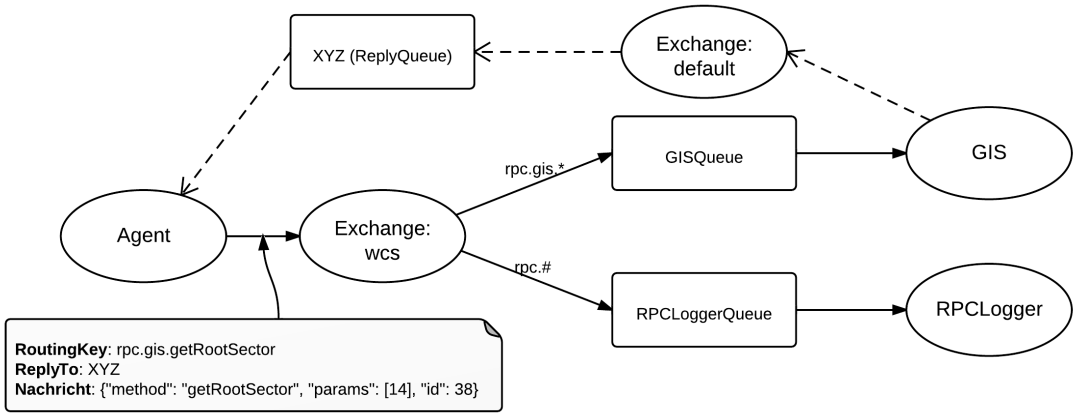

Abbildung 4.3.: Interne Struktur des WALK Communication Systems (WCS)

eine Komponente mit den an sie gerichteten Anfragen überfordert sein, kann schnell und unkompliziert eine weitere Instanz hinzugefügt werden, und die Last auf die beiden Instanzen aufgeteilt werden.

Diese Aspekte entsprechen den in 1.2 definierten Zielen:

- **Skalierbarkeit**: Durch den Einsatz von RabbitMQ skaliert das System sehr gut, da hohe Last einfach auf viele Instanzen einer Komponente verteilt werden kann.
- **Flexibilität**: Neue Komponenten können durch die sehr allgemeine Kommunikationsschnittstelle sehr einfach in das System eingebunden werden. Auch bestehende Komponenten können leicht ersetzt werden, da eine sehr lose Kopplung zwischen den Komponenten besteht.
- **Performance**: Wie in Kapitel 6 gezeigt wird, konnte die Leistung des Systems durch die Umstellung auf ein neues Kommunikationssystem deutlich gesteigert werden.
- **Zuverlässigkeit**: Durch die Zusicherung der Auslieferung von Nachrichten durch RabbitMQ ist es möglich, sehr zuverlässige Komponenten zu erstellen. Dass eine Nachricht auch tatsächlich ankommt muss nun nicht mehr der Sender prüfen, sondern wird vom Kommunikationssystem zugesichert.

4.4. Nachrichtenformat

4.4.1. Aufbau einer Nachricht

Die über das WCS versandten Nachtrichten sollten ein einheitliches Format haben, um den Aufwand bei Wartung und Erweiterung von bestehendem Code möglichst gering zu halten. Durch die Verwendung von Services (siehe Abschnitt 4.1) und Client-Bibliotheken (siehe Abschnitt 4.2) ist es zwar prinzipiell dem Entwickler eines Services überlassen, wie die Nachrichten erstellt und dann wieder interpretiert werden, ein einheitliches Format ist aber dennoch sehr sinnvoll.

Mit der *JavaScript Object Notation (JSON)* steht ein hochperformantes (vgl. Nurseitov u. a. (2009)) und sehr einfaches Nachrichtenformat zur Verfügung, das unter json.org sehr gut dokumentiert und in praktisch jeder Programmiersprache verfügbar ist. Für Remote Procedure Calls wird in WALK das Protokoll JSON-RPC (siehe json-rpc.org) verwendet. Es zeichnet sich besonders durch seine Einfachheit – die gesamte Spezifikation passt auf zwei DIN A4 Seiten – und Effizienz aus.

Listing 4.4.1 zeigt den Nachrichtenkörper einer RPC-Anfrage. Das Schlüsselwort *method* benennt den Methodennamen, *params* folgt eine Liste von Parametern und *id* ordnet jeder Anfrage eine entsprechende Antwort zu. Bei einem asynchronen RPC ist der Wert von *id* laut Spezifikation *null*. Die Kodierung von komplexen Objekten als Parameter oder Rückgabewerte wird in 4.4.3 vorgestellt.

```
1 { "method": "getRootSector", "params": [14], "id": 38 }
```

Auf die Anfrage eines synchronen RPC folgt immer eine Antwort. Listing 4.4.1 zeigt exemplarisch den Aufbau einer solchen Antwortnachricht.

```
1 { "result": {"id":"id15",...}, "error": null, "id": 38 }
```

4.4.2. RabbitMQ-Header

Um Nachrichten vom Sender zum Empfänger transportieren zu können, benötigt RabbitMQ einige Metadaten, die nicht im Nachrichtkörper auftauchen sollen. Dies hätte zwei wesentliche Nachteile: 1) würden die Nachrichten so unnötig komplex werden und 2) müsste RabbitMQ für das Routing einer Nachricht jedes mal den Nachrichtenkörper interpretieren. Dies würde zu einer wesentlich schlechteren Performance führen und es wäre nicht möglich, das Nachrichtenformat unabhängig vom Kommunikationssystem zu ändern.

Die Lösung ist es, bestimmte Header zu verwenden, die von RabbitMQ standardmäßig angeboten und jederzeit ausgelesen oder verändert werden können. Für den Einsatz in WALK sind drei Header besonders wichtig:

1. Der **Routingkey** legt fest, an welche Queues eine Nachricht gesendet werden soll. Bindings legen durch bestimmte Muster fest, für welche Nachrichten (mit welchem Routingkey) sich ein Konsument interessiert (vgl. Abschnitt 4.3).
2. Die **Correlation id** ordnet Nachrichten einer Session zu. So können zum Beispiel Antwortnachrichten einer bestimmten Anfrage zugeordnet werden.
3. Die **ReplyQueue** gibt an, an welche Queue die Antwortnachrichten zu einer bestimmten Anfrage gesendet werden sollen. Diese ist besonders bei synchronem RPC von großem Interesse.

Listing 4.4.2 zeigt das setzten dieser drei Werte in Java.

```
String corrId = java.util.UUID.randomUUID().toString();
String routingkey = "rpc.gis.getRootSector";

// creating message properties for replies
BasicProperties props = new BasicProperties
    .Builder()
    .correlationId(corrId)
    .replyTo(replyQueueName)
    .build();

// sending the request to the specified exchange,
// with a routingkey specifying the target operation
channel.basicPublish(EXCHANGE_NAME,routingkey,props,message);
```

4.4.3. Marshalling und Unmarshalling

Einige Operationen in WALK verlangen es, dass nicht nur primitive, sondern auch sehr komplexe Datentypen als Parameter und Rückgabewerte genutzt werden können. Ein Beispiel für eine solche Operation ist die Methode ‘‘`Sector getRootSector(String simulationId)`’’ der GIS-Komponente. Der Rückgabewert vom Typ *Sector* ist relativ komplex und muss in JSON übersetzt werden (marshalling), bevor er in eine Antwortnachricht eingefügt werden kann.

Die Logik zum Übersetzen von komplexen Objekten selbst zu entwickeln hat mehrere Nachteile:

- Es muss für jeden Datentyp ein eigener Marshaller und ein Unmarshaller geschrieben werden, was sehr viel Entwicklungsarbeit bedeutet.
- Jedes mal, wenn sich ein Datentyp ändert, müssen auch Marshaller und Unmarshaller angepasst werden.

- Es gibt sehr viele, sehr gute Bibliotheken, die diese Arbeit erledigen. Deren Qualität zu erreichen ist mit vertretbarem Aufwand nur schwer möglich.

In WALK wird für diese Arbeit die von Google entwickelte Gson-Bibliothek verwendet. Sie ist weit verbreitet und sehr einfach zu benutzen. Listing 4.4.3 zeigt, wie das *Sector-Objekt* mit Hilfe der Gson-Bibliothek kodiert und wieder dekodiert werden kann. Die Voraussetzung für das Dekodieren auf der Empfängerseite ist, dass eine entsprechende Klassendefinition vorhanden ist.

```
1  Vector3D pos = new Vector3D(1, 2, 3);
2  Vector3D bnd = new Vector3D(4, 4, 4);
3  Sector s = new Sector("id15",pos,bnd,null);
4
5  Gson gson = new Gson();
6  String Json = gson.toJson(s);
7  System.out.println(Json);
8
9  Sector s2 = gson.fromJson(Json, Sector.class);
10 System.out.println(s.equals(s2));
11
12 ------------------------------------------------
13 Ausgabe:
14 {"id":"id15","position":{"x":1.0,"y":2.0,"z":3.0},"bounds":{"x"
      :4.0,"y":4.0,"z":4.0},"children":[],"neighbours":[],"layers":{
      }}
15 true
```

In der Ausgabe ist zu erkennen, dass Gson eine sehr schlanke und für Menschen gut lesbare Repräsentation des Sektors erzeugt. Es ist zudem möglich, das kodierte Java-Objekt in jeder anderen Programmiersprache zu dekodieren – einen entsprechenden Unmarshaller vorausgesetzt.

4.5. Fehlerbehandlung

Tritt bei einer per RPC aufgerufenen Operation ein Fehler auf, dann muss dieser besonders behandelt werden. Im Normalfall wird einfach eine Exception an die aufrufende Funktion gereicht, die diese dann weiterreicht oder angemessen behandelt. Bei einem RPC muss die Meldung über den Fehler aber in eine Nachricht verpackt und an den Aufrufer gesendet werden.

JSON-RPC spezifiziert dazu das Schlüsselwort *error* in der Response-Nachricht. Als Wert kann error beliebige Daten enthalten, also auch eine kodierte Java-Exception. Ist bei einer Anfrage kein Fehler aufgetreten, muss der Wert von *error* 'null' sein.

Listing 4.5 zeigt ein Beispiel für eine Antwortnachricht mit kodiertem Exception-Objekt.

```
{ "result": null, "error": {"detailMessage":"Fehler␣in␣Operation␣
   xyz␣aufgetreten."}, "id": 38}
```

So ist eindeutig spezifiziert, wann eine Antwortnachricht als Fehlermeldung interpretiert werden muss, und wie dies geschehen kann. Eine weitere Möglichkeit wäre es, Fehlercodes global zu definieren und nur diese zu versenden. Das würde die zu versendenden Nachrichten zwar kleiner machen, gleichzeitig würde aber mehr Aufwand durch die Verwaltung der Fehlercodes anfallen.

4.6. Schnittstellen

Wenn komplexe Typen in Nachrichten übertragen werden sollen, muss erst ein Marshalling und später ein Unmarshalling dieser Daten stattfinden (vgl. 4.4.3). Dieser Vorgang ist sehr zeitaufwendig und kann zu erheblichen Verzögerungen der Simulation führen, wenn große Mengen von Objekten betroffen sind.

Daher ist es sehr sinnvoll, an allen externen Schnittstellen möglichst nur primitive Datentypen wie *String* und *Integer* zu verwenden. Diese können ohne Marshalling in Nachrichten eingebettet werden. So ist ein wesentlich schnellerer Ablauf der Simulation möglich.

Außerdem führt die Verwendung von sehr schlanken Schnittstellen zu einer losen Kopplung, da die Typen nur in den Komponenten bekannt sein müssen, in denen Sie auch verwendet werden. Das verringert wiederum den Wartungsaufwand bei Änderungen an den Datentypen, da sich diese nur auf einen relativ kleinen Teil des Systems auswirken.

5. Konzept zur Architekturdokumentation

Hauptziel der Architekturdokumentation ist es, allen Mitarbeitern die Möglichkeit zu geben, einen schnellen und unkomplizierten Einblick in die WALK-Architektur und deren technische Details zu erhalten. Außerdem soll die Qualität des Gesamtsystems durch eine saubere Dokumentation verbessert werden, da auch Themen wie Randbedingungen, technische Konzepte und Projektrisiken mit abgedeckt werden.

Die Dokumentation sollte immer parallel zum Gesamtsystem weiterentwickelt werden und befindet sich somit in einem stetigen Wandel. Daher ist es sinnvoll die Dokumentation (wie die Architektur selbst) iterativ zu entwickeln und immer einen benutzbaren Stand zu veröffentlichen.

Das hier vorgestellte Konzept zur Dokumentation von Software-Architektur orientiert sich hauptsächlich an dem in Starke und Hruschka (2002) vorgestellten arc42-Template. Die Erläuterungen zum diesem Template sind teilweise aus Starke (2011) entnommen.

Die folgenden Abschnitte beschreiben die Inhalte und Aufgaben der entsprechenden Abschnitte in der realen Dokumentation.

5.1. Abschnitt: Einführung und Ziele

Dieser Abschnitt beschreibt die wichtigsten Anforderungen an das System. Hier sollte kurz dargelegt werden, was das fertige System leisten soll.

Die **fachliche Aufgabenstellung** beschreibt den Grund, aus dem das System überhaupt entwickelt wird. Dieser kann zum Beispiel durch eine kurze Beschreibung der wichtigsten Anwendungsfälle untermauert werden.

Die **Qualitätsziele** beschreiben die wichtigsten nicht-funktionalen Anforderungen, die das System erfüllen soll. Hier können zum Beispiel Punkte wie Performance, Erweiterbarkeit, Skalierbarkeit, Bedienbarkeit oder ähnliches aufgeführt werden. Konkrete Zahlen sind hier oft nützlich:

> "Es soll möglich sein, 1000 Nachrichten pro Sekunde zwischen zwei Agentenplattformen auszutauschen, die sich auf unterschiedlichen Rechnern in einem lokalen 1GB-Netz befinden."

Ein Liste aller **Stakeholder** zeigt die wichtigsten Personen oder Rollen auf, die ein Interesse an der Entwicklung des Systems haben. Hier sollten auf jeden Fall der Name und die Kontaktdaten der Person, aber auch deren Ziele mit dem fertigen System aufgeführt werden.

> "Der Architekt Herr Müller möchte WALK einsetzen, um die Fluchtwege in von ihm geplanten Gebäuden zu verifizieren."

5.2. Abschnitt: Randbedingungen

Randbedingungen beeinflussen die Realisierung des Systems in einer bestimmten Weise. Es gibt viele Faktoren, die als architekturrelevante Randbedingungen gelten. Eine Schwierigkeit besteht darin, diese von den nicht relevanten Faktoren zu unterscheiden. Starke (2011) beschreibt drei Kategorien in die alle Einflussfaktoren aufgeteilt werden können:

- Organisatorische und politische Faktoren
- Technische Faktoren
- System- oder Produktfaktoren (alle funktionalen und nicht-funktionalen Eigenschaften eines Systems)

Einflussfaktoren können aber auch juristische Fragen sein, wie: "Welche rechtlichen Konsequenzen ergeben sich aus der Veröffentlichung der Simulationsergebnisse?" oder "Welche Konsequenzen hat der Einsatz von externen Bibliotheken oder Werkzeugen?"

5.3. Abschnitt: Kontextabgrenzung

> "Die Kontextabgrenzung zeigt das Umfeld eines Systems sowie dessen Zusammenhang mit seiner Umwelt." (Starke, 2011)

Die Kontextabgrenzung ist die Sicht auf das System auf der höchsten Abstraktionsebene. Sie zeigt das System nur als Blackbox und beschreibt alle Verbindungen zu den bekannten Nachbarsystemen sowie die ein- und ausgehenden Daten.

Die Beschreibung des Kontexts wird einmal auf fachlicher und einmal auf technischer Ebene durchgeführt. Auf der fachlichen Ebene sind besonders die Nachbarsysteme und die Schnittstellen zu diesen wichtig. Auf der technischen Ebene sollte das technische Umfeld in Form von spezifischen Hardwareeigenschaften und Kommunikationskanälen oder -protokollen beschrieben werden.

5.4. Abschnitt: Lösungsstrategie

Dieser Abschnitt beinhaltet drei sehr wichtige Punkte:

1. Eine kurze Beschreibung der wichtigsten Entscheidungen und Lösungsansätze (z.B. die Entscheidung für eine verteilte, komponentenorientierte Architektur).

2. Die Motivation, die zu den Hauptentscheidungen geführt hat ("Wir haben uns für den Einsatz einer Nachrichtenorientieren Middleware entschieden, weil...").

3. Eine Beschreibung der zentralen Entwurfsentscheidungen, die durch die Analyse der Kernaufgabe des Systems, die Qualitätsziele und Randbedingungen erarbeitet wurden. Der Übergang zwischen diesem und dem vorherigen Punkt ist fließend. Bei Bedarf können diese auch zusammengefasst werden.

Die Hauptaufgabe der Beschreibung der Lösungsstrategie ist es, dem Team und den Stakeholdern eine Idee davon zu vermitteln, warum der Architekt bestimmte Entscheidungen getroffen hat, um diese besser zu verstehen und bewerten zu können. Hier wird die Lösung des konkreten Problems (z.B. "Die Simulation von großen Menschenmengen") sehr abstrakt und allgemein geschildert, ohne im Detail auf technische Konzepte einzugehen.

5.5. Abschnitt: Bausteinsicht

> "Die Bausteinsicht bildet die Aufgaben des Systems auf Software-Bausteine oder -Komponenten ab. Diese Sicht macht Struktur und Zusammenhänge zwischen den Bausteinen der Architektur explizit. Bausteinsichten zeigen statische Aspekte von Systemen." (Starke, 2011)

Die Bausteinsicht hat mehrere Abstraktionsebenen, auf denen verschiedene Aspekte des Systems beschrieben werden. Jeder Baustein wird als Blackbox (nur externe Schnittstellen) und – eine Ebene tiefer – als Whitebox (Innensicht, ggf. mit weiteren Blackboxen) dargestellt. So kann sowohl die interne Struktur, als auch die Umgebung eines Bausteins beschrieben werden.

Abbildung 5.1 zeigt exemplarisch die Hierarchie von Blackboxen und Whiteboxen, die die WALK-Architektur auf drei Abstraktionsebenen beschreibt. Ebene 1 zeigt die Kontextabgrenzung des Gesamtsystems. Ebene 2 zeigt das System als Whitebox mit den beiden Komponenten *GIS* und *Common* als Blackboxen. Ebene 3 zeigt die interne Struktur dieser beiden Komponenten mit den enthaltenen Klassen.

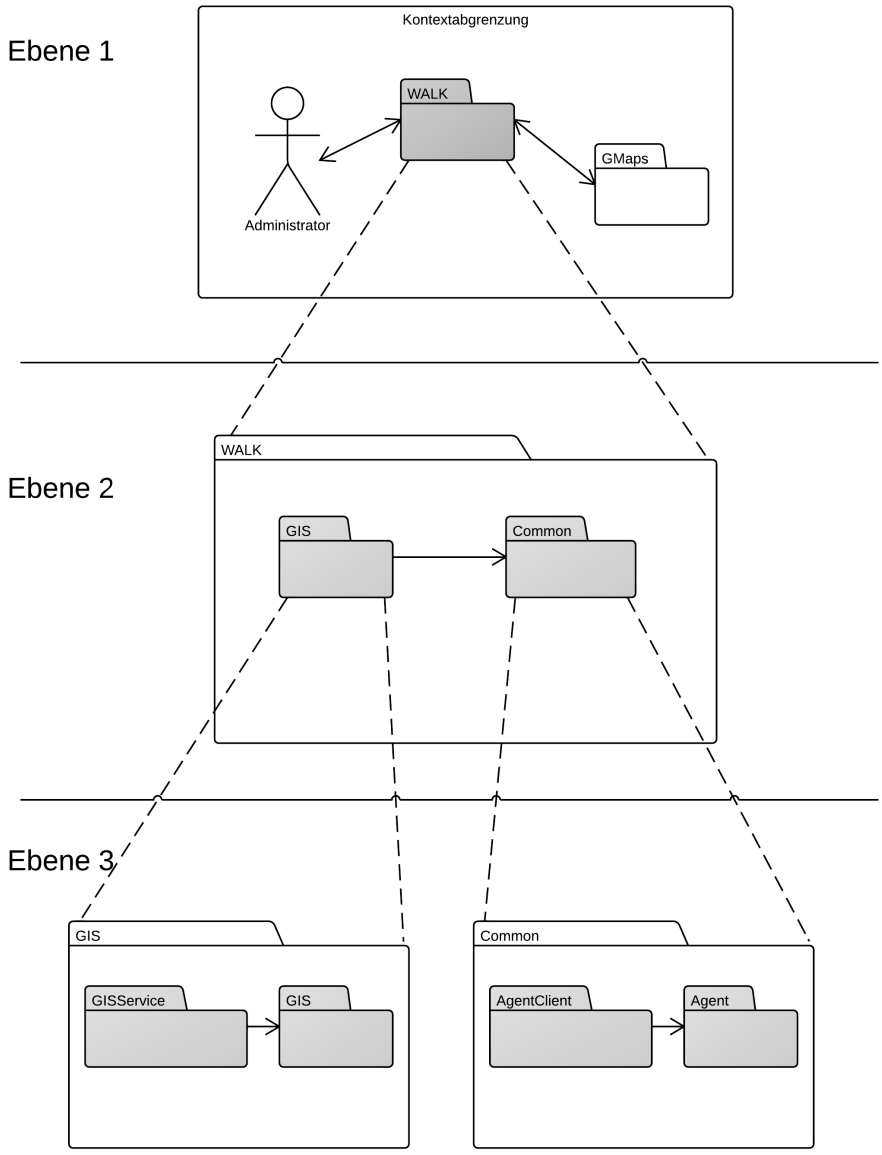

Abbildung 5.1.: Ein Beispiel der top-down-Verfeinerung der Bausteinsicht

Die Bausteinsicht liefert einen Überblick über die Struktur des Systems, die auf den tieferen Ebenen immer detaillierter wird. In dieser Sicht ist Einfachheit und Verständlichkeit sehr wichtig. Es sind also vereinfachte Darstellungen den komplexen aber vollständigen vorzuziehen. Es muss z.B. nicht jede Klasse aufgeführt werden, wenn dies nicht unbedingt für das Verständnis des Systems notwendig ist.

Starke (2011) definiert je ein Template zur Beschreibung eines Bausteins als Blackbox und als Whitebox, das unverändert in WALK übernommen wurde:

Blackbox-Template

1. Name
2. Zweck / Verantwortlichkeit
3. Schnittstellen
4. Ablageort / Datei
5. Erfüllte Anforderungen
6. Variabilität
7. Offene Punkte

Whitebox-Template

1. Name
2. Überblick (Diagramm)
3. Begründung
4. Enthaltene Blackboxes
5. Interne Schnittstellen
6. Offene Punkte

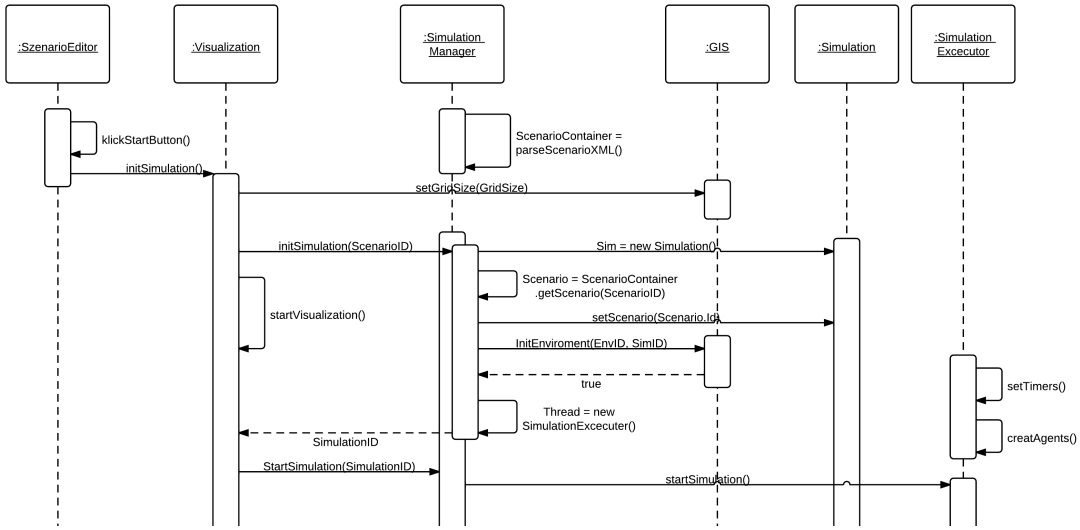

Abbildung 5.2.: Laufzeitsicht des Simulationsstarts in WALK

5.6. Abschnitt: Laufzeitsicht

Die Laufzeitsicht zeigt, welche Komponenten zur Laufzeit existieren und wie diese interagieren. Sie kann zum Beispiel einen komplexen Konfigurationsprozess einer Komponenten oder eines Subsystems zeigen. Hierfür werden meistens UML-Sequenzdiagramme oder -Aktivitätsdiagramme verwendet.

Komplexe Geschäftsprozesse können in der Laufzeitsicht dargestellt werden und zur Verifikation des fertigen Systems und zum Entwurf von Testszenarien dienen.

Abbildung 5.2 zeigt den Start einer Simulation. Hier sind die Nachrichten zu sehen, die – entsprechend ihrer zeitlichen Reihenfolge – von den beteiligten Komponenten ausgetauscht werden.

5.7. Abschnitt: Verteilungssicht

Die Verteilungssicht zeigt die Umgebung, in der das System abläuft. Hier werden die physikalischen Systeme und ihre Komponenten detailliert beschrieben. Zu nennen sind hier zum Beispiel: Prozessoren, Speicher, Netzwerkhardware oder physikalische Verbindungen.

Die Verteilungssicht ermöglicht es, Engpässe und Möglichkeiten der Hardware, auf der das System ausgeführt wird, rechtzeitig und korrekt zur erkennen. Sie bieten einen Überblick über die gesamte IT-Landschaft, die für das System relevant ist.

5.8. Abschnitt: Technische Konzepte

Zur Beschreibung der technischen Konzepte gehören Themen wie die verwendete Datenbank und das genutzte Messaging-System. Hier werden die eingesetzten Techniken im Detail beschrieben und Gründe für die Entscheidung für genau diese Technik angeführt. Das hilft anderen, die Entscheidung für eine bestimmte Technik nachzuvollziehen und zu akzeptieren.

Starke (2011) beschreibt ein Template zum Verfassen von technischen Konzepten:

1. Ziele und Anfoderungen
2. Randbedingungen
3. Kontext
4. Lösung / Vorgehen
    a) Strukturen / Abläufe
    b) Beispiele inkl. Code
5. Alternativen
6. Risiken

In WALK kann hier z.B. das Messaging, die Verteilung des Systems, der Einsatz von Datenbanken und die Realisierung der verschiedenen grafischen Oberflächen beschrieben werden. Aber auch andere Konzepte wie Fehlerbehandlung oder Sicherheit sollten hier erwähnt werden.

5.9. Abschnitt: Entwurfsentscheidungen

Hier werden die Entwurfentscheidungen beschrieben, die das Gesamtsystem (also die Architektur) betreffen. Genauso wichtig wie die getroffenen Entscheidungen, sind die verworfenen Alternativen. Dadurch ist jedem Leser klar, ob eine bestimmte Alternative schon untersucht wurde und aus welchem Grund sie verworfen wurde. So wird weniger Arbeit doppelt gemacht und gewonnene Erkenntnisse werden dokumentiert.

Das in WALK eingesetzte und aus Starke (2011) übernommene Template umfasst die folgenden Punkte:

1. Was muss entschieden werden (Fragestellung)?
    a) In welchem Kontext?

2. Wie wurde entschieden?
    a) Warum?
    b) Getroffene Annahmen?
    c) Verworfene Alternativen?
3. Konsequenzen?
4. Bekannte Risiken?
5. Wer hat wann entschieden?

5.10. Dokumentation der WALK-Architektur

Die WALK-Architektur wird im Projekt-Wiki (siehe WALK) dokumentiert. Die Verwendung eines Wikis erlaubt allen Beteiligten einen schnellen und unkomplizierten Zugriff auf die Dokumentation. Diese Erleichterung des Zugriffs soll dazu führen, dass jede Architekturerweiterung möglichst zeitnah dokumentiert wird und so die implementierte und die dokumentierte Architektur möglichst gering auseinander driften.

6. Evaluation

In den folgenden Abschnitten wird versucht, einen Teil des Nutzens dieser Arbeit in Zahlen auszudrücken. Dazu wird ein Versuch durchgeführt, der die Nachrichtenlaufzeiten des alten Kommunikationssystems mit denen des neuen vergleicht. In Abschnitt 6.1 wird der Aufbau des Versuchs beschrieben. Es folgt dann in Abschnitt 6.2 eine Beschreibung der Durchführung des Versuchs und in Abschnitt 6.3 die Interpretation der Ergebnisse. Abschnitt 6.4 geht auf einige Einschränkungen der erarbeiteten Lösung ein.

6.1. Aufbau

Der Versuch besteht aus zwei Szenarien, die die Leistung der beiden unterschiedlichen Arten von Kommunikation in WALK messen. Szenario 1 führt einen synchronen RPC aus und misst die Zeit, die vom Aufruf bis zum Eintreffen der Antwort vergeht. Szenario 2 simuliert das Auftreten eines Ereignisses und misst die Dauer des anschließenden asynchronen Nachrichtenaustauschs zwischen Publisher und Subscriber.

Um eine möglichst realistische Testumgebung zu schaffen, wurden die benötigten WALK-Komponenten auf zwei verschiedene Computer verteilt. Auf der Maschine *evalclient* wird eine Client-Anwendung ausgeführt, die Anfragen an einen bestimmten WALK-Dienst sendet. Auf der Maschine *simmng* läuft der Simulationsmanager, der diese Anfragen entgegen nimmt, bearbeitet und gegebenenfalls beantwortet. Beide Testcomputer haben eine minimale Ubuntu 11.10 Server Installation und sind über einen Gigabyte-Switch in einem dedizierten Netzwerk miteinander verbunden.

Abbildung 6.1 zeigt die Verteilung der Komponenten auf die beiden Maschinen.

6.2. Durchführung

In jedem Szenario wird zuerst über das alte Kommunikationssystem (*walk-communication.jar*) kommuniziert und anschließend das neue WALK Communication System (WCS) benutzt. Es werden die Paketlaufzeiten nach unterschiedlichen Kriterien und auf unterschiedliche Weise gemessen.

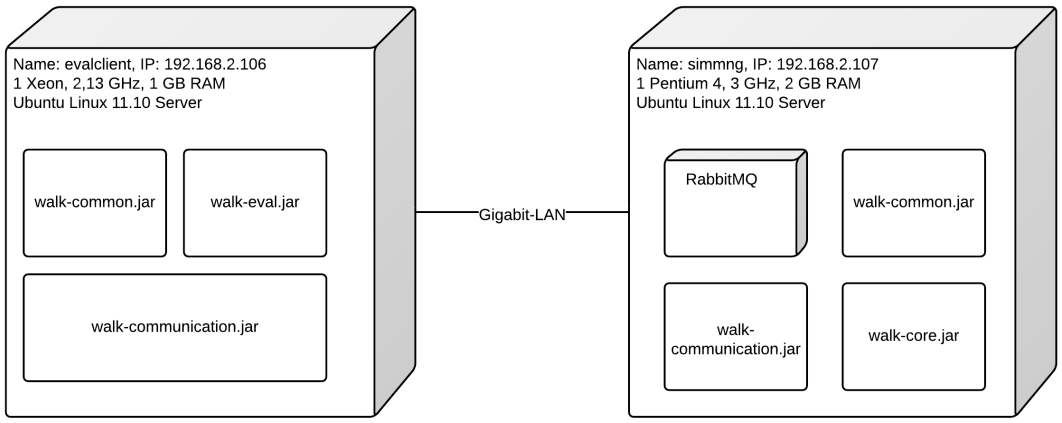

Abbildung 6.1.: Verteilung der Artefakte des Versuchsaufbaus auf die physischen Systeme

In beiden Szenarien wird die Messung für 1, 10, 100, 500, 1000, 5000 und 10000 gesendete Nachrichten jeweils zehnmal durchgeführt. Aus diesen zehn Messwerten wird dann der Mittelwert $\overline{x}$ gebildet: $\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$, wobei n die Anzahl der Messwerte und x_i ein einzelner Messwert ist.

Um die Signifikanz der Ergebnisse zu zeigen wird außerdem die Standardabweichung σ nach der folgenden Formel angegeben: $\sigma = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \overline{x})^2}$

In den folgenden beiden Unterabschnitten wird die Durchführung der Testszenarien im Detail beschrieben.

6.2.1. Szenario 1: Synchrone Kommunikation

In diesem Szenario wird die Operation "`List<AgentType> getAgentTypes()`" des *SimulationManagers* durch einen synchronen RPC unterschiedlich oft aufgerufen. Es wird die Zeit in Millisekunden gemessen, die vom Aufruf dieser Operation bis zum Vorliegen des Ergebnisses auf dem System *evalclient* verstreicht.

Zur Messung der Zeit wird die Java-Funktion `java.lang.System.currentTimeMillis()` verwendet, die die aktuelle Systemzeit in Millisekunden ausgibt. Listing 6.2.1 zeigt den Programmcode zur Messung der Laufzeit der Nachrichten über das alte Kommunikationssystem.

```
1 start = System.currentTimeMillis();
2 SimulationManager sm = scs.getSimulationManager();
3 for(int i=0; i<times;i++)
4     types = sm.getAgentTypes();
5 long runtime = System.currentTimeMillis() - start;
```

Tabelle 6.1.: Messdaten der Performanceuntersuchung für Szenario 1

Anz. Nachrichten	**altes System**	σ **(in %)**	**neues System**	σ (in %)
1	244 ms	4 ms (1,63)	34 ms	0 ms (0)
10	376 ms	6 ms (1,6)	88 ms	1 ms (1,14)
100	1467 ms	22 ms (1,5)	582 ms	10 ms (1,72)
500	5420 ms	92 ms (1,7)	2643 ms	29 ms (1,1)
1000	10027 ms	180 ms (1,8)	5261 ms	66 ms (1,25)
5000	43767 ms	544 ms (1,24)	26537 ms	461 ms (1,74)
10000	84546 ms	1056 ms (1,25)	58252 ms	968 ms (1,66)

Legende: σ = Standardabweichung

Tabelle 6.1 zeigt die in Szenario 1 gemessenen Werte für das alte und für das neue System. In Abbildung 6.2 werden diese grafisch dargestellt.

6.2.2. Szenario 2: Asynchrone Kommunikation

Im zweiten Szenario wird von einen *Publisher* ein Event ausgelöst und an einen *Subscriber* (hier: SimulationManager) gesendet. Da die Kommunikation asynchron erfolgt, kann nicht wie in Szenario 1 die Zeit zwischen dem Aufruf und dem Eintreffen der Antwort gemessen werden. Um trotzdem eine aussagekräftige Zeitmessung vornehmen zu können, wird diese im Subscriber durchgeführt. Der Publisher sendet drei unterschiedliche Arten von Nachrichten an den Subscriber: 1) eine Start-Nachricht, nach deren Erhalt er mit der Zeitmessung beginnt, 2) viele Test-Nachrichten, die einfach nur gezählt werden und 3) ein Ende-Nachricht, nach deren Erhalt die Zeitmessung beendet und das Ergebnis ausgegeben wird.

Auch in diesem Szenario wird die Funktion `java.lang.System.currentTimeMillis()` zur Zeitmessung verwendet. Den zur Zeitmessung des alten Systems verwendeten Programmcode zeigt Listing 6.2.2.

```
1  // Publisher-Code
2  System.out.println("Starte␣asynchrone␣Messung␣mit␣\"alt\"");
3  scs.publish(startMessage);
4  for(int i=0; i<times;i++)
5      scs.publish(testMessage);
6  scs.publish(endMessage);
7
8  // Subscriber-Code
9  ChannelMessageProcessor start = new ChannelMessageProcessor() {
10     public void processMessage(ChannelMessage channelMessage) {
11         starttime = System.currentTimeMillis(); }};
```

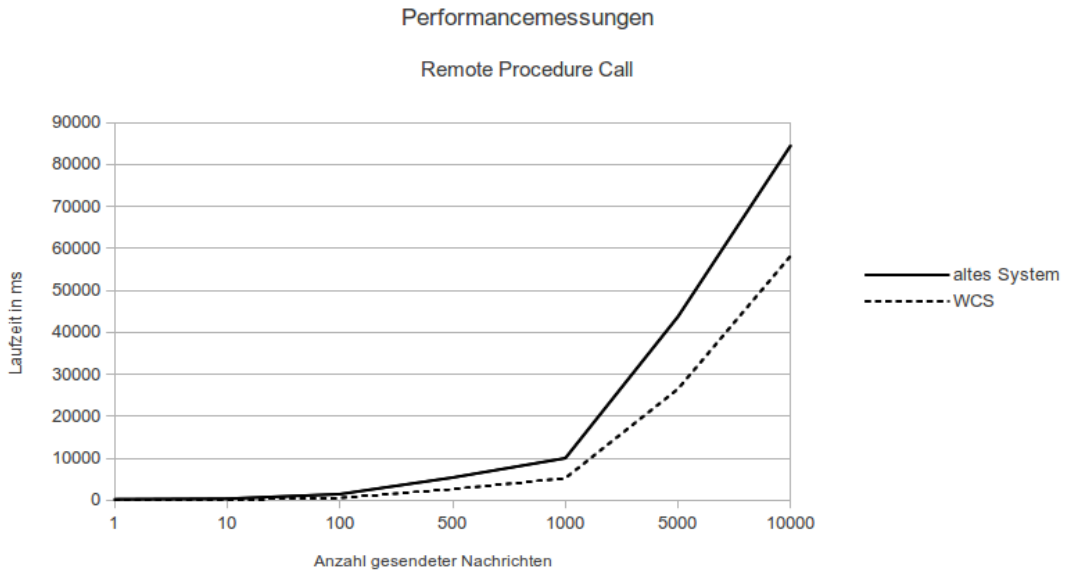

Abbildung 6.2.: Grafische Darstellung der Nachrichtenlaufzeiten beim Versenden RPCs

```
12
13 ChannelMessageProcessor end = new ChannelMessageProcessor() {
14     public void processMessage(ChannelMessage channelMessage) {
15         runtime = System.currentTimeMillis() - starttime;}};
16
17 ChannelMessageProcessor cmp = new ChannelMessageProcessor() {
18     public void processMessage(ChannelMessage channelMessage) {
        msg++ }};
19
20 this.communicationSystem.subscribe("StartChannel", start);
21 this.communicationSystem.subscribe("EndChannel", end);
22 this.communicationSystem.subscribe("TestChannel", cmp);
```

Tabelle 6.2 zeigt die in Szenario 2 gemessen Werte. In Abbildung 6.3 werden diese grafisch dargestellt.

Der gesamte Programmcode, der zur Erfassung der Messdaten verwendet wurde, befindet sich in Anhang C.

Tabelle 6.2.: Messdaten der Performanceuntersuchung für Szenario 2

Anz. Nachrichten	**altes System**	σ **(in %)**	**neues System**	σ (in %)
1	28 ms	1 ms (3,57)	1 ms	0 ms (0)
10	41 ms	0 ms (0)	4 ms	0 ms (0)
100	193 ms	3 ms (1,55)	34 ms	0 ms (0)
500	568 ms	11 ms (1,94)	110 ms	2 ms (1,82)
1000	1010 ms	15 ms (1,49)	195 ms	3 ms (1,54)
5000	4606 ms	55 ms (1,19)	982 ms	13 ms (1,32)
10000	9617 ms	157 ms (1,63)	1919 ms	33 ms (1,72)

Legende: σ = Standardabweichung

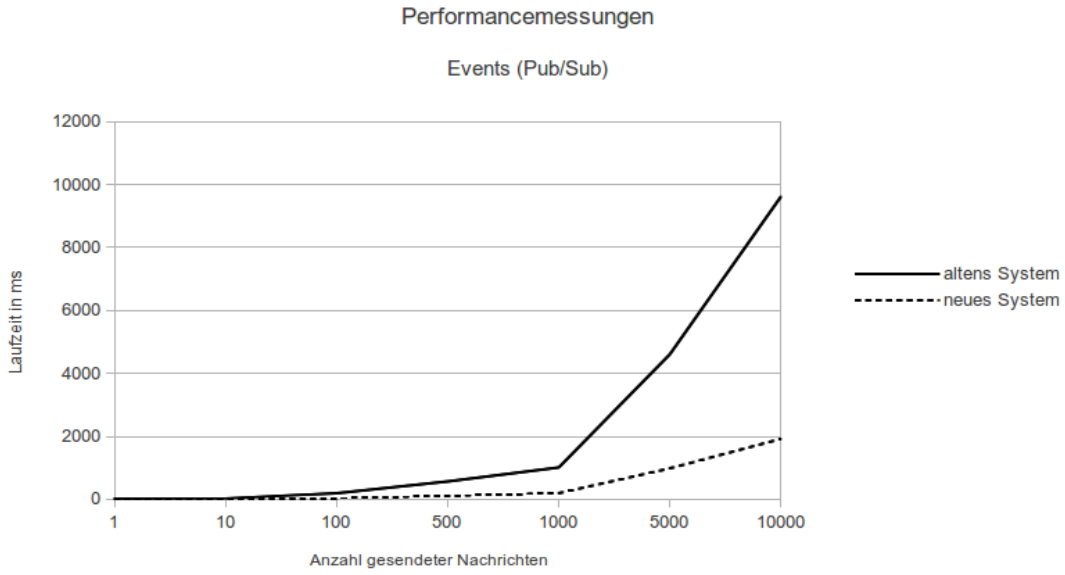

Abbildung 6.3.: Grafische Darstellung der Nachrichtenlaufzeiten beim Versenden von Events

6.3. Interpretation

Den Messdaten ist eindeutig zu entnehmen, dass die Umstellung des Kommunikationssystems eine signifikante Verbesserung der Performance des Simulationssystems gebracht hat. Die Testimplementierung des Clients und des Services ist nur rudimentär erfolgt und bietet noch einiges Potenzial für zusätzliche Leistungssteigerungen in der Nachrichtenübermittlung und -verarbeitung.

Die gemessene Leistungssteigerung ist somit hauptsächlich auf den Einsatz von RabbitMQ statt des bisherigen Kommunikationssystems und JSON in Verbindung mit Gson statt XML und eines selbst entwickelten Marshallers/Unmarshallers zurückzuführen. Weitere Optimierungsmöglichkeiten werden in Kapitel 7 aufgeführt.

6.4. Einschränkungen und Randbedingungen

Eine der wichtigsten Einschränkungen, die durch den Einsatz des neuen Kommunikationssystems entstehen, ist die Bindung an das Produkt RabbitMQ. Dieses muss in der Infrastruktur des Simulationssystem vorhanden und sauber konfiguriert sein. Es wird also entsprechendes Expertenwissen auf diesem Gebiet benötigt.

Sollte eine zukünftige Version von RabbitMQ nicht mehr unter der MPL, sondern einer restriktiveren Lizenz erscheinen, würde das einen Einsatz in WALK erschweren. Allerdings erlaubt es die Architektur von WALK auch das Kommunikationssystem ohne großen Aufwand auszutauschen und so auf ein anderes Produkt umzusteigen.

Weiterhin ist zu beachten, dass jedes Team, das für eine Komponente mit öffentlichem Service verantwortlich ist, auch die benötigten Clients zur Verfügung stellen muss. Wie diese Regelung im weiteren Verlauf des Projekts gehandhabt wird, sollte im Einzelfall entschieden werden. Es ist durchaus denkbar, dass ein Client in einer speziellen Technologie, auch von dem Benutzer des Services in Zusammenarbeit mit dem Anbieter erstellt wird.

7. Zusammenfassung und Ausblick

Dieses Kapitel liefert eine kurze Zusammenfassung dieser Arbeit und gibt einen Ausblick auf mögliche Erweiterungen und Verbesserungen der erarbeiteten Lösung.

7.1. Zusammenfassung

In dieser Arbeit wurde zum Einen ein Dokumentationskonzept für die Software-Architektur des WALK-Simulationssystems entwickelt und zum Anderen dessen Kommunikationsstruktur neu gestaltet.

Ein Vergleich verschiedener Messaging-Software-Produkte in Abschnitt 3.3 hat ergeben, dass die nachrichtenbasierte Middleware RabbitMQ sich besonders gut für den Einsatz in WALK eignet. Durch den Austausch der selbst entwickelten Middleware durch RabbitMQ konnten die Nachrichtenlaufzeiten messbar reduziert werden (siehe Kapitel 6).

Nach der Analyse verschiedener Konzepte zur Dokumentation von Software-Architektur in Abschnitt 3.4, wurde das arc42-Template ausgewählt und für die Dokumentation von WALK modifiziert. Im Rahmen dieser Arbeit wurde ein Template erstellt, dass den WALK-Mitarbeitern hilft, die von ihnen entwickelten Softwarekomponenten sinnvoll zu dokumentieren.

7.2. Ausblick

Da sich WALK als Forschungsprojekt in einen stetigen Wandel befindet und oft neue Technologien ausprobiert werden, ist auch die Architektur ständig mit neuen Anforderungen konfrontiert. Es wird also notwendig sein, die Architektur ständig den sich ändernden Anforderungen anzupassen.

In Zukunft ist es sinnvoll, die Weiterentwicklung von ZeroMQ im Auge zu behalten und eine spätere, stabilere Version erneut zu testen. Das Konzept der verteilten Message Queues scheint ein viel versprechender Ansatz zu sein, der die Performance der Simulation noch einmal steigern könnte. Dies ist hauptsächlich darauf zurückzuführen, dass lokale Aufrufe zwar an die Message Queue gesendet werden, nicht aber über das Netzwerk geleitet werden, da die Message Queue auf jedem Rechner lokal ausgeführt wird.

Auch ist der Einsatz eines kompakteren Nachrichtenformats wie zum Beispiel MessagePack (siehe Furuhashi Sadayuki) in Betracht zu ziehen. Dabei ist besonders die wesentlich kleinere Nachrichtengröße und damit auch geringere Übertragungszeit interessant. Auch hier ist vermutlich noch eine Verkürzung der Simulationsdauer möglich.

Des Weiteren steht eine Untersuchung der Testbarkeit des Systems und die Entwicklung einer Testfallstruktur für das Kommunikationssystem noch aus. Im Rahmen dieser Arbeit konnten diese Themen leider nicht behandelt werden. Sie sind aber von großer Bedeutung für die Qualität des Gesamtsystems und daher ein interessanter Ansatz für weiterführende Arbeiten.

A. Client und Client-Library

SimulationManagerClient (client library)

```
public class SimulationManagerClient implements SimulationManager
{
    private static final String BASIC_ROUTINGKEY = "rpc.simmng";
    private static final String REMOTE_HOST = "10.0.0.2";
    private final MQClient mqclient;

    public SimulationManagerClient()
    {
        mqclient = new MQClient(REMOTE_HOST);
    }

    /**
    * Sends a synchronous remote procedure call with no
    * parameter to the SimulationManager Service.
    *
    * @return List of available AgentTypes
    */
    @Override
    public List<AgentType> getAgentTypes()
    {
        String methodName = "getAgentTypes";
        List<Object> params = new ArrayList<Object>();
        return mqclient.call(BASIC_ROUTINGKEY,methodName,params);
    }
}
```

Verwendung der client library

```
SimulationManagerClient smc = new SimulationManagerClient();
List<AgentType> types = smc.getAgentTypes();
```

B. MQClient

```
/**
 * @version 0.1 2012/02/27
 * @author Till Kahlbrock <till.kahlbrock@haw-hamburg.de>
 *
 * The MQClient connects the services and the clients by serving
 * several operation for synchronous and asynchronous
    communication.
 * It hides the technical messaging code from the services and
    the clients
 * by providing a clean interface with no mq-specific operations.
 *
 */
public class MQClient
{
    private static final String EXCHANGE_NAME = "wcs";
    private ConnectionFactory factory;
    private Channel channel;
    private Connection connection;

    public MQClient(String mqhost)
    {
        factory = new ConnectionFactory();
        factory.setHost(mqhost);
        connection = factory.newConnection();
        channel = connection.createChannel();
    }

    /**
     * Subscribes a consumer to the given topic and injects the
     * consumer to a given callback object. The callback may
     * handle messages on this topic.
     *
```

```
31      * @param topic
32      * @param callback
33      */
34     public void subscribe(String topic, Callback callback)
35     {
36         // create channel
37         Connection connection = factory.newConnection();
38         Channel channel = connection.createChannel();
39 
40         //create exchange
41         channel.exchangeDeclare(EXCHANGE_NAME, "topic");
42 
43         // create queue and bind it to the exchange
44         String queueName = channel.queueDeclare().getQueue();
45         channel.queueBind(queueName, EXCHANGE_NAME, topic);
46 
47         // create and subscribe consumer
48         QueueingConsumer consumer = new QueueingConsumer(channel)
               ;
49         channel.basicConsume(queueName, true, consumer);
50 
51         // inject the consumer and run the callback
52         callback.setConsumer(consumer);
53         callback.start();
54     }
55 
56     /**
57      * Publishes a message to a given topic.
58      *
59      * @param topic
60      * @param message
61      */
62     public void publish(String topic, String message)
63     {
64         boolean mandatory = true;
65         boolean immediate = true;
66         channel.basicPublish(EXCHANGE_NAME, topic, mandatory,
               immediate, null, message.getBytes());
67     }
68 
```

```
69      /**
70       * Sends a response to the specified reply queue with a given
             correlation id.
71       *
72       * @param response The response message
73       * @param corrId The correlation id
74       * @param replyTo The queue to send reply to
75       */
76      public void reply(String response, String corrId, String
            replyTo)
77      {
78          BasicProperties replyProps = new BasicProperties
79              .Builder()
80              .correlationId(corrId)
81              .build();
82          channel.basicPublish("", replyTo, replyProps, response.
                getBytes());
83      }
84
85      /**
86       * Makes a synchronous remote procedure call by building a
             json-rpc
87       * message and sending it to the exchange specified by
88       * <code>basicRoutingKey +"."+ methodName</code>
89       *
90       * @param basicRoutingKey
91       * @param methodName
92       * @param params
93       * @return return value of type T
94       */
95      public <T> T call(String basicRoutingKey, String methodName,
            List<Object> params)
96      {
97          Gson gson = new Gson();
98          String routingKey = basicRoutingKey +"."+ methodName;
99          String corrId = java.util.UUID.randomUUID().toString();
100         String jsonParams = params.isEmpty() ? "null" : gson.
                toJson(params);
```

```
101         String msg = String.format("{\"method\" : \"%s\", \"
                params\":%s, \"id\": %s}",methodName,jsonParams,corrId
                );
102
103         byte[] message = msg.getBytes();
104         T response = null;
105
106         String replyQueueName = channel.queueDeclare().getQueue()
                ;
107         QueueingConsumer consumer = new QueueingConsumer(channel)
                ;
108         channel.basicConsume(replyQueueName, true, consumer);
109
110         // creating message properties for replies
111         BasicProperties props = new BasicProperties
112             .Builder()
113             .correlationId(corrId)
114             .replyTo(replyQueueName)
115             .build();
116
117         // sending the request to the specified exchange,
118         // with a routingkey specifying the target operation
119         channel.basicPublish(EXCHANGE_NAME,routingKey,props,
                message);
120
121         // waiting for a response (BLOCKING!)
122         while (true) {
123             QueueingConsumer.Delivery delivery = consumer.
                    nextDelivery();
124             if (delivery.getProperties().getCorrelationId().
                    equals(corrId)) {
125                 String json = new String(delivery.getBody());
126                 Type type = new TypeToken<T>(){}.getType();
127                 response = gson.fromJson(json, type);
128                 break;
129             }
130         }
131     }
132 }
```

C. WCSTimer

```
1  public class WCSTimer
2  {
3      public long measure(String system, int times) throws
           Exception
4      {
5          long runtime = 0;
6          long start   = 0;
7          List<AgentType> types = new ArrayList<AgentType>();
8
9          // Synchrone Messung des alten Systems
10         if(system.equals("old"))
11         {
12             System.out.println("Starte␣synchrone␣Messung␣mit␣\"
                   alt\"");
13             start = System.currentTimeMillis();
14             Properties walkProperties = new Properties();
15             PropertyConfigurator.configure("./config/log4j.
                   properties");
16             walkProperties.load(new FileInputStream("./config/
                   walk.properties"));
17             SimpleCommunicationSystem scs = new
                   SimpleCommunicationSystem(walkProperties);
18             SimulationManager sm = scs.getSimulationManager();
19             for(int i=0; i<times;i++)
20                 types = sm.getAgentTypes();
21             runtime = System.currentTimeMillis() - start;
22             scs.shutdown();
23             System.out.println("Fertig");
24         }
25         // Synchrone Messung des neuen Systems
26         else
27         {
```

```
28              System.out.println("Starte␣synchrone␣Messung␣mit␣\"
                    neu\"");
29              start = System.currentTimeMillis();
30              SimulationManagerClient smc = new
                    SimulationManagerClient();
31              for(int i=0; i<times;i++)
32                  types = smc.getAgentTypes();
33              runtime = System.currentTimeMillis() - start;
34
35              System.out.println("Fertig");
36          }
37          if(types.isEmpty())
38              throw new Exception("Ergebnis␣nicht␣korrekt!");
39          return runtime;
40      }
41
42      public void measureAsync(String system, int times) throws
            Exception
43      {
44          // Asynchrone Messung des alten Systems
45          if(system.equals("old"))
46          {
47              System.out.println("Starte␣asynchrone␣Messung␣mit␣\"
                    alt\"");
48              Properties walkProperties = new Properties();
49              PropertyConfigurator.configure("./config/log4j.
                    properties");
50              walkProperties.load(new FileInputStream("./config/
                    walk.properties"));
51              SimpleCommunicationSystem scs = new
                    SimpleCommunicationSystem(walkProperties);
52              ChannelMessage startMessage = new
                    SimulationDataMessage("StartChannel", "start",
                    SimulationDataType.Update, 1221, new ArrayList<
                    SimulationObject>());
53              ChannelMessage endMessage = new SimulationDataMessage
                    ("EndChannel", "end", SimulationDataType.Update,
                    1222, new ArrayList<SimulationObject>());
54              ChannelMessage testMessage = new
                    SimulationDataMessage("TestChannel", "test",
```

```
                SimulationDataType.Update, 1223, new ArrayList<
                SimulationObject>());
55              scs.publish(startMessage);
56              for(int i=0; i<times;i++)
57                  scs.publish(testMessage);
58              scs.publish(endMessage);
59              scs.shutdown();
60              System.out.println("Fertig");
61          }
62          // Asynchrone Messung des neuen Systems
63          else
64          {
65              System.out.println("Starte asynchrone Messung mit \"
                    neu\"");
66              MQClient mqclient = new MQClient("wcs");
67              mqclient.publish("event.gis.start", "start");
68              for(int i=0; i<times; i++)
69                  mqclient.publish("event.gis.test", "test");
70              mqclient.publish("event.gis.end", "end");
71              System.out.println("Fertig");
72          }
73      }
74  }
```

Abbildungsverzeichnis

Literaturverzeichnis

[AMQP Working Group 2008] AMQP Working Group: *AMQP - Advanced Message Queuing Protocol - Protocol Specification v. 0-9-1*. AMQP Working Group (Veranst.), November 2008

[Apache Software Foundation] Apache Software Foundation: *Apache ActiveMQ*. – URL `http://activemq.apache.org/amqp.html`. – Letzter Zugriff: 08.03.2012

[Baldowski 2011] Baldowski, M: *Entwicklung eines 3D-Geoinformationssystem für Gefahrensituationen im In- und Outdoorbereich (Projektbericht SoSe 2011)*. 2011

[Bass u. a. 2003] Bass, Len ; Clements, Paul ; Kazman, Rick: *SEI series in software engineering*. Bd. 2nd: *Software Architecture in Practice*. Addison-Wesley Professional, 2003. – 560 S. – ISBN 0321154959

[Birrell und Nelson 1984] Birrell, Andrew D. ; Nelson, Bruce J.: Implementing remote procedure calls. In: *ACM Transactions on Computer Systems* 2 (1984), Nr. 1, S. 39–59. – URL `http://portal.acm.org/citation.cfm?doid=2080.357392`. – ISBN 0897911156

[Carnegie Mellon Software Engineering Institute] Carnegie Mellon Software Engineering Institute: *Software Architecture Glossary*. – URL `http://www.sei.cmu.edu/architecture/start/glossary/definitions.cfm`. – Letzter Zugriff: 28.03.2012

[Clements u. a. 2010] Clements, Paul ; Bachmann, Felix ; Bass, Len ; Garlan, David ; Ivers, James ; Little, Reed ; Merson, Paulo ; Nord, Robert: *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2010 (SEI series in software engineering). – 592 S. – ISBN 0321552687

[Furuhashi Sadayuki] Furuhashi Sadayuki: *MessagePack*. – URL `http://msgpack.org`. – Letzter Zugriff: 28.03.2012

[Henjes 2010] Henjes, R: *Performance Evaluation of Publish/Subscribe Middleware Architectures*. Würzburg, Julius-Maximilians-Universität Würzburg, Dissertation, 2010. – 90–94 S

[Hohpe und Woolf 2004] Hohpe, G ; Woolf, B: *Enterprise integration patterns: designing, building, and deploying messaging solutions*. Addison-Wesley, 2004 (The Addison-Wesley signature series). – ISBN 9780321200686

[json-rpc.org] json-rpc.org: *JSON-RPC Specifications*. – URL `http://json-rpc.org/wiki/specification`. – Letzter Zugriff: 28.03.2012

[json.org] json.org: *JSON Projektseite*. – URL `http://json.org`. – Letzter Zugriff: 28.03.2012

[Kruchten 1995] Kruchten, Phillippe: Architecture blueprints—the "4+1" view model of software architecture. In: *IEEE Software* 12 (1995), Nr. 6, S. 42–50. – URL `http://portal.acm.org/citation.cfm?doid=216591.216611`

[Münchow 2012] Münchow, Stefan: *Ausarbeitung Anwendungen 1 WiSe 2011/12*. 2012

[Nurseitov u. a. 2009] Nurseitov, Nurzhan ; Paulson, Michael ; Reynolds, Randall ; Izurieta, Clemente: Comparison of JSON and XML Data Interchange Formats: A Case Study. In: *Scenario* 59715 (2009), S. 157–162. – URL `http://www.cs.montana.edu/izurieta/pubs/caine2009.pdf`

[Second Life Wiki 2010] Second Life Wiki: *Message Queue Evaluation Notes — Second Life Wiki*. 2010. – URL `http://wiki.secondlife.com/w/index.php?title=Message_Queue_Evaluation_Notes&oldid=702853`. – Letzter Zugriff: 08.03.2012

[Springsource] Springsource: *RabbitMQ Plugins*. – URL `http://www.rabbitmq.com/plugins.html`. – Letzter Zugriff: 28.03.2012

[Springsource] Springsource: *Springsource Webpage*. – URL `http://www.springsource.com`. – Letzter Zugriff: 08.03.2012

[Starke 2011] Starke, G: *Effektive Software-Architekturen: Ein praktischer Leitfaden*. Hanser Fachbuchverlag, 2011. – ISBN 9783446427280

[Starke und Hruschka 2002] Starke, G ; Hruschka, P: *Das arc42 Template*. 2002. – URL `http://www.arc42.de/template/template.html`. – Letzter Zugriff: 08.03.2012

[Sustrik 2007] Sustrik, Martin: *Messaging enabled network*. 2007. – URL `http://www.zeromq.org/whitepapers:messaging-enabled-network`. – Letzter Zugriff: 08.03.2012

[Sustrik 2008] Sustrik, Martin: *Broker vs. Brokerless.* 2008. – URL `http://www.zeromq.org/whitepapers:brokerless`

[Tanenbaum und Van Steen 2007] Tanenbaum, A S. ; Van Steen, Maarten: *Distributed Systems: Principles and Paradigms, 2/E.* Prentice Hall, 2007. – 686 S. – ISBN 9780132392273

[The Open Group 2009] The Open Group ; Ed, Th (Hrsg.): *TOGAF Version 9.* The Open Group, 2009. – 778 S. – ISBN 9789087532307

[Thiel 2011] Thiel, C: *Eine Plattform für Fußgängersimulationen (Projektbericht SoSe 2011).* 2011

[Thiel-Clemen u. a. 2011] Thiel-Clemen, T ; Köster, G ; Sarstedt, S: Walk - Emotion-based pedestrian movement simulation in evacuation scenarios. In: Wittmann, J et al. (Hrsg.): *Simulation in den Umwelt- und Geowissenschaften.* Berlin : Shaker, 2011

[WALK] WALK: *Walk-Wiki.* – URL `http://walk.informatik.haw-hamburg.de/wiki/index.php/Architektur_Template`. – Letzter Zugriff: 28.03.2012

[Wooldrige 2002] Wooldrige, M: *An Introduction to MultiAgent Systems.* Chichester, England : John Wiley and Sons, 2002

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 29. März 2012 Till Kahlbrock