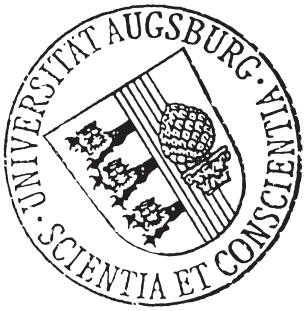
Vergleich aktueller Implementierungen von Message Oriented Middlewares

Magnus Görlitz

Thesis
for the degree of
Bachelor of Science (B.Sc.)



University of Augsburg
Department of Computer Science
Software Methodologies for Distributed Systems

February 2019

Vergleich aktueller Implementierungen von Message Oriented Middlewares

Erstprüfer:	**Prof. Dr. Bernhard L. Bauer**, Department of Computer Science, University of Augsburg, Germany
Zweitprüfer:	**Prof. Dr. Bernhard Möller**, Department of Computer Science, University of Augsburg, Germany
Betreuer:	**M.Sc. Melanie Langemeier**, Department of Computer Science, University of Augsburg, Germany
Firmenbetreuer:	**M.Sc. Fabian Zintgraf**, Team Lead IT, CHECK24 Versicherungsservice GmbH, Germany

Kurzbeschreibung

In der modernen Softwareentwicklung finden sich in großen Anwendungen häufig sogenannten *Message Oriented Middlewares* wieder. Hierbei handelt es sich um Systeme, die eine effiziente Kommunikation zwischen einer großen Anzahl von Prozessen ermöglichen, oft auch in verteilten Systemen oder heterogenen Netzen. In den letzten Jahren gab es vermehrt Weiterentwicklungen im Bereich der Message Oriented Middleware, da diese nicht zuletzt auch in Microservices vermehrte Verbreitung gefunden haben. Hierbei handelt es sich um Message Oriented Middlewares, die als eigene Anwendung unabhängig vom restlichen System betrieben werden. Es existieren viele Implementierungen, wobei sich diese untereinander stark unterscheiden können. Bei der Konzeption eines Systems, welches eine Message Oriented Middleware verwendet, verliert man daher schnell den Überblick über diese.

Aus diesem Grund wird diese Arbeit einen Überblick über bestehende Lösungen geben und diese auf konzeptioneller Ebene vergleichen. Einzelne Punkte, in denen sich diese unterscheiden, werden dabei begründet erarbeitet und zu einem Katalog zusammengefasst werden. Dies ermöglicht, die Erkenntnisse aus der Arbeit auch auf zukünftige Entwicklungen anzuwenden. Daraufhin wird ein Vergleich erstellt, um die aktuell führenden Implementierungen aufzeigen und ihre Unterschiede in den Punkten dieses Kataloges gegenüberzustellen. Letztendlich soll für ein Zielsystem des Betreuers (CHECK24 Versicherungsservice GmbH) eine geeignete Implementierung gewählt werden. Da die Umsetzung den Rahmen der Arbeit sprengen würde, wird sich hier auf eine Analyse der möglichst optimalen Lösungen beschränkt. Hierzu werden für dieses System spezifische Anforderungen identifiziert und versucht, mit dem im zweiten Teil der Arbeit vorgestellten Katalog eine Kategorie für dieses Zielsystem zu finden. Daraufhin kann eine Empfehlung einer Lösung ausgesprochen werden. Da eine konkrete Implementierung entfällt, werden theoretische Probleme der empfohlenen Implementierung erörtert und Lösungsvorschläge präsentiert werden.

Zusammenfassend wird zum einen ein Überblick über Implementierungen erstellt, der diese konkret vergleicht. Dabei werden einzelne Punkte des Vergleichs vorher erarbeitet und anschließend nach Relevanz eingeordnet. Letztendlich werden anhand eines Fallbeispiels die vorherigen Abschnitte validiert und ein Fazit gezogen.

Danksagung

An dieser Stelle möchte ich meinem Firmenbetreuer Fabian Zintgraf danken, der maßgeblich zur Inspiration der Arbeit beigetragen und mich stets unterstützt hat. Der Austausch über die Inhalte meiner Arbeit mit ihm und dem Team des CHECK24 Versicherungscenters waren von unschätzbarem Wert.

Inhaltsverzeichnis

1

Einführung

1.1 Motivation

In der CHECK24 Versicherungsservice GmbH kommen im *Versicherungscenter*[1] im Laufe des Jahres neue Businessanforderungen auf die vorhandene Infrastruktur zu. Im Rahmen der Neuerungen wurde erwogen, eine Message Oriented Middleware in das System zu integrieren. Dabei reichten jedoch die Erfahrungen anderer Teams nicht aus, um eine Entscheidung bezüglich der konkreten Implementierungen zu fällen.

Die konkreten Anforderungen waren zu unterschiedlich zu denen von anderen Teams, um von Ihren Erfahrungen sinnvolle Schlüsse ableiten zu können. Daher soll eine Evaluierung der Optionen stattfinden, was den Anreiz für diese Arbeit gab. Es soll also nicht nur eine Analyse und ein Vergleich von marktführenden Lösungen stattfinden, sondern auch in einem Praxisteil das konkrete Szenario untersucht und eine Empfehlung ausgesprochen werden.

Der bis dahin aufgestellte Eigenschaftenkatalog und die Analyse existierender Implementierungen sollen helfen, eine fundierte Wahl einer Lösung zu treffen. Zudem sollen die erarbeiteten Grundlagen Missverständnisse vorbeugen und zum Verständnis bei der Integration beitragen.

[1] https://www.check24.de/versicherungscenter/, abgerufen am 23.11.2018

1.2 Problemstellung

Um einen möglichst universellen Vergleich zwischen Message Oriented Middlewares zu ermöglichen, muss der Begriff zuerst umfangreich definiert werden. Nur auf dieser Grundlage ist eine Gegenüberstellung von Implementierungen sinnvoll.

Weiter soll der Vergleich aufgrund ausgewählter, wichtiger Eigenschaften von Message Oriented Middlewares geschehen. Diese müssen also erarbeitet und anhand Ihrer Relevanz eingeschätzt werden. Dabei können Eigenschaften beliebig detailliert oder abstrakt sein. Es besteht also eine Herausforderung darin, ein geeignetes Abstraktionslevel zu finden. Anschließend soll dieser Eigenschaftenkatalog anhand eines Praxisteils evaluiert werden.

Eine weitere Herausforderung besteht im Vergleich bestehender Message Oriented Middlewares. Es sollte aus den verbreiteten Lösungen eine repräsentative Menge ausgewählt werden. Also eine Menge, die einerseits groß genug ist, um genug verschiedene Repräsentanten zu beinhalten, andererseits aber klein genug ist, um noch übersichtlich und relevant zu sein.

Somit sollten die Vergleichspunkte fundiert erarbeitet und nach Relevanz bewertet werden. Letztendlich sollte der Anspruch sein, eine realistisch nutzbare Orientierung bei der Planung eines Systems zu erstellen. Dementsprechend sind die grundlegenden Herausforderungen dieser Arbeit die Erarbeitung eines Eigenschaftenkatalogs und die Analyse einzelner ausgewählter Implementierungen damit, sowie die Anwendung und Evaluierung anhand eines Praxisbeispiels.

2

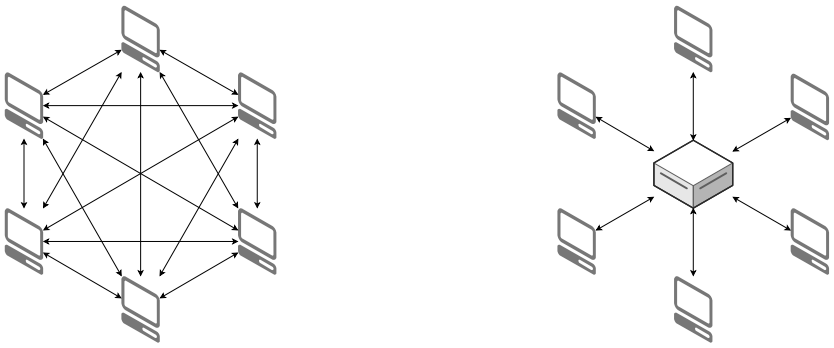
Grundlagen

2.1 Definition

2.1.1 Message Oriented Middleware

Eine *Message Oriented Middleware* (kurz MOM), also eine nachrichtenorientierte Middleware, wird definiert als eine beliebige Middleware, die es ihren *Clients* ermöglicht, Nachrichten aneinander zu versenden [18]. Diese entkoppelt Sender und Empfänger (in diesem Kontext auch *Producer* und *Consumer* genannt [7]), da nun nicht mehr alle Teilnehmer untereinander verbunden sein müssen, sondern nur noch mit der Middleware. Dies wird in den Darstellungen 2.1 abgebildet. Hierbei wird in beiden Fällen die Netzwerkarchitektur gezeigt, die benötigt wird, damit jeder Teilnehmer an jeden anderen Teilnehmer Nachrichten senden kann. Dargestellt sind Clients bzw. bei (b) Clients und in der Mitte eine MOM.

Dabei beschreibt der Begriff Message Oriented Middleware keine konkrete Umsetzung. Eine solche besteht in den meisten modernen Implementierungen aus einem oder mehreren Brokern, die nach einem gewissen Schema routen sowie Queues, die Nachrichten speichern können. Diese Strukturen werden in den nächsten Abschnitten detailliert erklärt. [4]



(a) Aufbau ohne MOM (b) Aufbau mit MOM

Abbildung 2.1: Netzwerkarchitektur bei Verwendung direkter Kommunikation im Vergleich mit einer Message Oriented Middleware

Ein *Message Broker* sorgt für das Weiterleiten der Nachrichten an den richtigen Empfänger. Dabei kann er auch Nachrichten, die in einem bestimmten Format angenommen wurden, in ein anderes, zum Empfänger kompatibles Format umwandeln. Er sorgt also für eine Entkopplung von Nachrichtenformaten. Bei der

Verwendung eines Brokers werden alle Nachrichten über diesen geleitet. Er verarbeitet die Nachrichten und leitet sie nach gewissen Regeln weiter. Man spricht auch von einem *Routing*, hierauf wird in Abschnitt 2.1.2 näher eingegangen. [18]

Eine Queue ist im Kontext von MOM eine Speicherlösung für Nachrichten. Dabei können je nach Implementierung eine oder mehrere Queues für die gesamte MOM existieren. Auch ob eine Queue vor den Message Broker geschaltet wird, oder dieser seine Nachrichten an eine Queue weiterleitet, ist implementierungsabhängig [2, 14]. Warum Queues verwendet werden und welche Vorteile dies haben kann, wird in Abschnitt 2.1.3 besprochen. Eine Veranschaulichung eines exemplarischen Gesamtaufbaus einer MOM mit den eben besprochenen Teilen ist in Darstellung 2.2 zu finden. Hierbei ordnet der Message Broker die empfangenen Nachrichten in eine der Queues ein, von denen die Consumer diese erhalten.

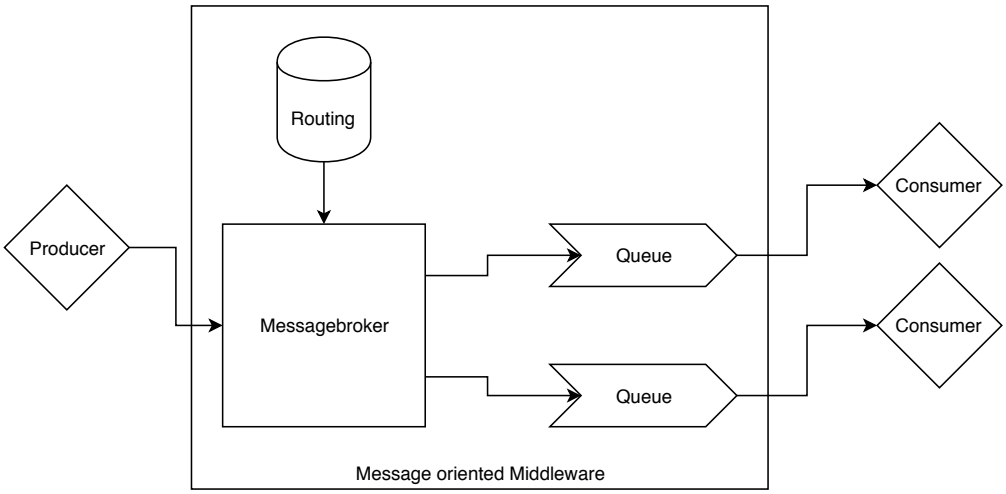


Abbildung 2.2: Exemplarischer Aufbau einer Message Oriented Middleware

Ein großer Vorteil, den eine MOM bieten kann, ist *asynchrone* Kommunikation. Ist eine Architektur synchron, so wird bei dem Verschicken der Nachricht der weitere Ablauf des Senders blockiert, bis eine Antwort erhalten und der Ablauf fortgesetzt wird. Bei der asynchronen Variante kann eine Nachricht verschickt werden, ohne dass der Ablauf dadurch blockiert wird. Dies ist vor allem in Batchverarbeitungs- und parallelen Systemen nützlich [18]. Zur Veranschaulichung dient Abbildung 2.3, in der der asynchrone Versand einer Nachricht dargestellt ist. Dabei wird die erste Nachricht von *Client 2* verarbeitet und danach eine Antwort verschickt. Weitere Vorteile werden in Abschnitt 2.1.4 besprochen.

In den untersuchten Implementierungen findet dabei das *Publish-Subscribe* Pattern Anwendung. Es beschreibt die Art, wie sich zwei verschiedene Teile einer Software miteinander verbinden und kommunizieren. Dabei ist bei Publish-Subscribe (kurz auch PubSub) eine Nachricht nicht an einen Empfänger direkt gerichtet, sondern Sender können Nachrichten veröffentlichen (publish) und interessierte Empfänger können ihr Interesse durch eine Subscription bekunden. Daraufhin erhalten alle Subscriber, beispielsweise eines bestimmten Themas, alle Nachrichten davon. Auf welche Art diese Auswahl stattfindet, wird genauer im nächsten Abschnitt (2.1.2 Routing) beschrieben. [7]

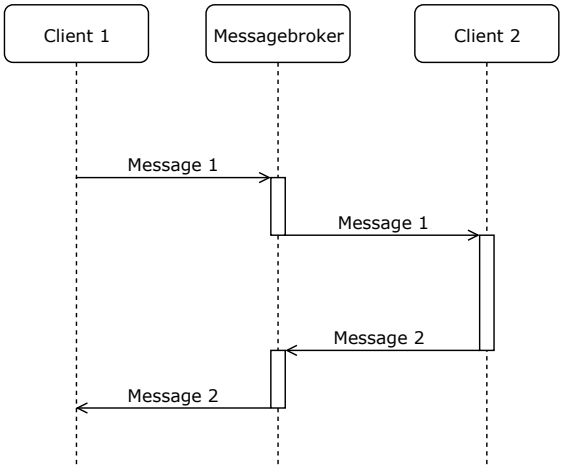


Abbildung 2.3: Asynchrone Kommunikation mithilfe eines Message Brokers

2.1.2 Routing

Message Broker müssen architekturbedingt ein Regelwerk anbieten, nach dem Nachrichten bestimmten Empfängern zugestellt werden, da die Nachrichten nicht direkt an diese geschickt werden. Man spricht von einem *Routing*. So kann man neben statisch festgelegten Routen auch *Channel-*, *Topic-*, oder *Content-*basiert geroutet werden [18]. Bei channelbasiertem Routing werden Kanäle gebildet, denen Clients beitreten können; daraufhin erhalten diese alle in dem Kanal veröffentlichten Nachrichten. Eine weitere Möglichkeit ist das Bilden von *Topics*, wodurch dann je nach thematischer *Subscription* des einzelnen Clients Nachrichten zugestellt werden [7]. Bei inhaltsbasiertem Routing wird basierend auf dem Inhalt der einzelnen Messages vom Broker beurteilt, welche Consumer diese erhalten [18].

Dabei sind je nach Protokoll und Implementierung verschiedene Ausprägungen anzutreffen. Die meisten bieten mindestens eine Topic-Logik, oft auch eine Bildung von Channels an. Sie unterscheiden sich hauptsächlich darin, wie mit mehreren Topics umgegangen wird. Es können beispielsweise Wildcards verwendet oder logische Verknüpfungen erstellt werden. Wenn z.B. gewünscht ist, dass ein Consumer alle Topics der Nachricht abonniert haben muss, um diese zu erhalten, kann dies unter anderem mit RabbitMQ bzw. dem verwendeten AMQP-Protokoll realisiert werden [5]. Weitere Details zu den jeweiligen Routingoptionen werden im Abschnitt 3 für die einzelnen Implementierungen besprochen.

2.1.3 Queues

Um eine stärkere Entkopplung zu erreichen, werden in modernen Implementierungen häufig Queues verwendet. Im Kontext von Message Oriented Middleware sind hier mit Queues Speicher für Nachrichten gemeint [4]. Je nach Implementierung wird eine Nachricht hier nur zwischengespeichert, bis sie erfolgreich weitergeleitet wurde, oder dauerhaft gespeichert. Dies hat zwei Vorteile. Zum einen können bei einer langsamen Abarbeitung von den Consumern die Nachrichten zwischengespeichert werden, und gehen nicht verloren. Zum anderen kann eine langfristige Speicherung beispielsweise als Backup fungieren. Dabei stößt man auf die üblichen Prinzipien von Warteschlangen. Im Kontext von nachrichtenorientierten Middlewares ist eine Queue meistens ein *FIFO*, gibt also die Nachrichten nach dem Zwischenspeichern in unveränderter Reihenfolge wieder an den Empfänger aus [4]. Es existieren jedoch Lösungen wie Kafka, bei denen zwar eine Ordnung innerhalb einzelner Queues vorliegt, jedoch die Broker zusätzlich zu ihrer normalen Publish-Subscribe Kommunikation auch gezielt beliebige gespeicherte Nachrichten abrufen können. Hierbei kann also nicht nach üblichen Kriterien von Warteschlangen bewertet werden [1]. Wie bereits erwähnt, ist es abhängig von der Implementierung, wie die Queue mit dem Rest der MOM zusammenarbeitet. Details hierzu werden im Abschnitt 3 für die jeweilige Implementierung besprochen.

2.1.4 Protokolle

Je nach Implementierung werden verschiedene Protokolle verwendet und unterstützt. Dabei existieren komplett proprietäre Protokolle bzw. Eigenentwicklungen, wie zum Beispiel bei ZeroMQ oder Kafka [1]. Abseits hiervon hat sich vorerst vor allem *JMS*, das Java Message Service Protokoll, durchgesetzt. Dies wird in ActiveMQ, JBOSS Messaging und einigen anderen Implementierungen verwendet [5]. Als populäres und standardisiertes Protokoll ist auch *AMQP*, Advanced Message Queueing Protocol, hervorzuheben [19]. Es findet in RabbitMQ, Qpid, HornetMQ und Weiteren Verwendung [5]. Daneben gibt es noch weitere Protokolle wie NMS, STOMP und WebSocket, allerdings wird in der Industrie hauptsächlich AMQP verwendet. [19] Erwähnenswert ist jedoch noch MQTT (Message Queue Telemetry Transport), ein im IoT-Bereich weit verbreitetes Protokoll. Es zeichnet sich durch seine Leichtgewichtigkeit und einfache Implementierung aus [5].

AMQP ist ein offener Standard, um Nachrichten zwischen zwei Anwendungen auszutauschen. Es wurde durch eine Arbeitsgruppe zusammengestellt und Anfang 2010 in seiner ersten Version präsentiert. Um von der großen Verbreitung von JMS profitieren zu können, wurde für Kompatibilität gesorgt [19]. Somit kann AMQP auch in einer JMS-Umgebung integriert werden, um von AMQPs erhöhter Kompatibilität mit weiterer Software profitieren zu können. AMQP legt fest, wie die Kommunikation zwischen den Clients und den Brokern abläuft, jedoch nicht, was der Broker tut. Daher können verschiedene Broker untereinander unterschiedliche Funktionalitäten zur Verfügung stellen und trotzdem AMQP unterstützen.

Die meisten Protokolle sind Netzwerkprotokolle und arbeiten auf der Anwendungsebene. Sie umfassen dabei Spezifizierungen darüber, wie die Nachrichten verschickt werden. Im Falle von AMQP wurde dies mit einem binären Peer-to-Peer Protokoll umgesetzt. Weiter wird meistens ein bestimmtes Nachrichtenformat vorgegeben, nach denen alle in der MOM propagierten Nachrichten strukturiert sind. Hierbei liegen auch die Unterschiede in Protokollen, da sie erstens noch weitere Teile des Kommunikationsprozesses standardisieren, und zweitens unterschiedliche Spezifizierungen für Übertragung und Nachrichtenformat mitbringen können. [19]

2.2 Vorteile

Durch die beschriebene Architektur einer nachrichtenorientierten Middleware ergeben sich einige Vorteile gegenüber direktem Messaging. Neben den bereits erwähnten sind dies die Folgenden.

Skalierbarkeit Durch die Extraktion des Messagings in eine externe Instanz kann nicht nur das Routing effizienter, simpler und performanter geschehen, es können auch weitere Empfänger und Sender einfacher eingebunden werden und die Netzwerktopologie wird vereinfacht. Weiter können bei den meisten Implementierungen Redundanz herbeigeführt und somit höhere Lasten bewältigt werden. Auf diese Weise kann beim Erweitern der Anwendung sehr einfach die nachrichtenorientierte Middleware mit skaliert werden [4].

Flexibilität Da die meisten Message Broker Clients für viele Programmiersprachen anbieten und auf mehreren Plattformen betrieben werden können, wird eine große Flexibilität erzielt. Dabei müssen die Empfänger und Sender nicht zwingend das gleiche Protokoll verwenden [5].

Stabilität Gerade durch Verwendung von Queues kann eine erhöhte Stabilität erreicht werden. Sollte ein Sender ausfallen, ist dies für die weitere asynchrone Verarbeitung der Nachricht nicht relevant, solange diese vorher an den Broker übermittelt wurde. Auch der Empfänger kann ausfallen, solange die Queue die Nachricht lange genug vorhalten kann. Je nach Implementierung ist sogar der Broker ausfallsicher, da durch Mechanismen wie Redundanz eine Unabhängigkeit von der Hardware erreicht wird [5].

Kohäsion und Kopplung Da bei direkter Kommunikation jeweils ein Client den anderen direkt aufruft, sind die zwei Systeme eng miteinander verknüpft. Sollten sich Teile ändern oder wegfallen, so müssen diese Änderungen durch das komplette Netzwerk propagiert werden. Aufgrund dieses sehr statischen Aufbaus liegt eine hohe Kopplung vor. Durch die Verwendung einer MOM wird die Aufgabe des Messagings komplett auf eine eigene Instanz verschoben. Somit wird eine enge Kohäsion und eine geringe Kopplung erzielt [4].

2.3 Probleme und Herausforderungen

Die Nutzung einer MOM bringt jedoch nicht nur Vorteile mit sich. Es gibt einige Punkte, die man bei der Planung berücksichtigen sollte.

Sicherheit Zwar sind viele Vorteile mit der Kommunikation über eine zentrale Instanz verbunden, allerdings ist damit auch jeder andere Teil der Anwendung davon abhängig. Fällt ein Broker bzw. die gesamte MOM aus, so kann dies beispielsweise dazu führen, dass gesendete und in der Queue vorgehaltene Nachrichten evtl. komplett verloren gehen. Dabei können also durch Ausfall eines einzigen Systems Daten des gesamten Systems verloren gehen. Ebenso muss für die Sicherheit eines weiteren Systems gesorgt werden. Kann ein Angreifer die Kontrolle über eine MOM übernehmen, hat er potentiell Zugriff auf alle Nachrichten, die über sie versendet werden. [18]

Betrieb Mit einer MOM muss ein weiteres System regelmäßig gewartet und gepflegt werden. Außerdem müssen Themen wie High Availability und Skalierung bedacht werden. Zwar liefern die meisten Implementierungen Wege mit, diesen Themen zu begegnen, allerdings geschieht dies selten automatisch [17]. Wenn unvorhersehbar eine große Nachrichtenmenge bei der MOM eingeht, ist ein reibungsloser Betrieb meist trotzdem wünschenswert. Damit Ausfälle bzw. Performanceengpässe überhaupt rechtzeitig bemerkt werden können, muss zusätzlich ein geeignetes Monitoring betrieben werden. [5]

Integration Jedoch ist der Betrieb nicht die einzige Herausforderung. So muss beispielsweise für jeden Teilnehmer eine geeignete Schnittstelle zum Broker erstellt werden. Auch wenn die meisten Broker *Libraries* in verschiedenen Sprachen zu Verfügung stellen, muss dies trotzdem erst einmal integriert werden [2]. Sollte die MOM nicht bereits bei der Konzeption eines Systems eingeplant worden sein, so entstehen Einführungskosten.

2.4 Typische Anwendungsszenarien

Es lassen sich einige Anwendungsbeispiele zusammentragen, die besonders von der Verwendung einer MOM profitieren.

Microservice-Architekturen Unter einem *Microservice* versteht man einen unabhängigen Prozess, der in einer *Microservice-Architektur* eine einzelne bestimmte Aufgabe erfüllt. Dabei ist eine Microservice-Architektur eine verteilte Anwendung, welche ausschließlich aus Microservices besteht [9]. Es müssen also viele kleinere Anwendungen miteinander kommunizieren. Hier kann mit einer MOM beispielsweise erreicht werden, dass nicht jeder Microservice die Adressen seiner Kommunikationspartner kennen muss. Das erleichtert unter anderem die Integration neuer Microservices.

IoT Auch im Bereich *Internet of Things*, in dem viele leistungsschwache Geräte permanent mit dem Internet verbunden sind, sind MOMs von Vorteil. Soll zwischen vielen Geräten untereinander kommuniziert werden, greifen oben genannte Vorteile [16]. Da dieser Trend gerade in den letzten Jahren Einzug in viele Privathaushalte findet, hat das Thema MOM auch eine wirtschaftliche Relevanz. Dabei kann unter anderem von der erhöhten Skalierbarkeit der Kommunikation mit einer MOM im Vergleich zur direkten Kommunikation profitiert werden.

Event Sourcing Event Sourcing beschreibt einen Ansatz, wie Datensätze, die häufigen Änderungen ausgesetzt sind, programmatisch behandelt werden können. Hierbei wird für jede Änderung ein sogenanntes *Event* erstellt, welches die Änderungen zum vorherigen Event enthält. Dabei ist es meist sinnvoll, wenn andere Teilanwendungen diese Events erhalten oder diese zumindest zentral abgespeichert werden. Grundsätzlich ist Event Sourcing im Kontext von Messaging nicht strikt von Microservices bzw. allgemein aufgetrennten Anwendungen trennbar, allerdings ist dieses Pattern in den letzten Jahren so populär geworden, dass speziell dafür geeignete Broker existieren [8, 1]. Event Sourcing kann hier besonders von MOMs mit einer Queue, die permanente Speicherung anbietet, profitieren, da hier alle Events abgelegt werden können.

3

Vergleich

3.1 Eigenschaften einer nachrichtenorientierten Middleware

Eigenschaft	Optionen
Standardisierte Protokolle	AMQP, MQPP, -
Clientbibliotheken	siehe 3.1
Möglichkeit zur Skalierung	Replikation, -
Durchsatz	hoch, niedrig
Latenz	stabil, nicht stabil
Routingoptionen	Topic, Channel, Content, Statisch
Zustellgarantien	at most once, at least once, exactly once
Speicherort	RAM, Festplatte
Langzeitspeicherung	unterstützt, nicht unterstützt
Ordnungsgarantien	total order, partitioned order, no order

Tabelle 3.1: Eigenschaftenkatalog

Nach den Erkenntnissen des vorherigen Abschnitts können nun Eigenschaften einer MOM erarbeitet werden, die besonders zur Unterscheidung von Implementierungen geeignet sind. Hierbei wird anhand des im Abschnitt 2.1 beschriebenen Aufbaus vorgegangen. Die erarbeiteten Eigenschaften sind in der Tabelle 3.1 zusammen mit den für die jeweiligen Eigenschaften verfügbaren Optionen dargestellt. Im Folgenden werden die einzelnen Punkte erklärt. Anschließend werden sie den folgenden Abschnitten dazu verwendet werden, einzelne Implementierungen zu analysieren und miteinander zu vergleichen.

3.1.1 Sender und Empfänger

Wie in den Grundlagen erwähnt, sind Producer und Consumer unabhängige Instanzen. Es ist jedoch nicht ausgeschlossen, dass Consumer gleichzeitig Producer sind und umgekehrt. Somit fließt in die Kriterien für die Auswahl von Brokern ein, welche Clients dieser unterstützen kann. Dabei muss sowohl betrachtet werden, ob **standardisierte Protokolle** verwendet werden, als auch, wie leicht Clients in verbreiteten Sprachen implementiert werden können. Nach Statistiken von Github sind die 2018 am meisten verwendeten Sprachen in absteigender Reihenfolge Javascript, Java, Python, PHP, C++, C#, Typescript, Shell, C und Ruby [10]. Da Typescript auch Javascript Pakete verwenden kann und somit in diesem Kontext die Unterscheidung zwischen Javascript und Typescript

irrelevant ist, sowie Shell für die meisten großen Softwarearchitekturen nicht infrage kommt, werden nur die anderen acht Sprachen zur Bewertung herangezogen. Dabei entfällt eine Bewertung der unterstützten Plattformen, da diese eher von den **Clientbibliotheken** bzw. den unterstützten Programmiersprachen abhängen. Diese sind in Tabelle 3.1 dargestellt.

In einer Implementierung ist selbstverständlich auch die Zahl an gleichzeitig unterstützten Clients nicht unlimitiert. Sie hängt von mehreren Faktoren ab, allerdings kann man hierbei durchaus an die Grenzen einer Hardware stoßen. In den meisten Fällen wird daher Replikation angeboten [1, 14]. Das bedeutet, dass einige oder alle Daten der MOM in mehreren, unabhängigen Instanzen gespeichert und synchronisiert werden. Somit werden Implementierungen auf **Möglichkeit zur Skalierung** untersucht.

Ebenso limitiert ist eine Implementierung darin, wie viele Nachrichten sie gleichzeitig versenden und empfangen kann, da hier Faktoren wie Hardwareressourcen und Netzwerk eine Rolle spielen. Relevante Metriken sind hierbei zum einen der **Durchsatz**, also die Menge von Nachrichten, die eine MOM pro Zeiteinheit verarbeiten kann, und zum anderen die **Latenz**, also die Versanddauer einer Nachricht. Diese Eigenschaften können beispielsweise in Echtzeitsystemen harte Anforderungen sein.

3.1.2 Routing

Wie bereits in Abschnitt 2.1.2 erwähnt, unterscheidet man neben statischem Routing zwischen *Channel-*, *Topic-* und *Content-* basiertem Routing. Daher werden diese **Routingoptionen** in den Eigenschaftenkatalog aufgenommen.

Beim Versand können jedoch Netzwerkpakete verloren gehen, Clients oder Broker ausfallen, oder Antwortpakete verloren gehen, wodurch die Nachricht mehrfach oder gar nicht gesendet wird. Daher sollen Implementierungen auf **Zustellgarantien** untersucht werden.

Bei **at most once** wird eine Nachricht maximal einmal zugestellt. Hierbei wird nicht berücksichtigt, ob eine Nachricht z.b. durch Paketverlust verloren geht. Eine weitere Variante ist **at least once**. Hierbei wird jede Nachricht garantiert zugestellt, allerdings kann es zu Mehrfachzustellungen kommen und es werden mehr Ressourcen benötigt. Die dritte Alternative **exactly once**, bei dem keine

Broker/Sprache	Javascript	Java	Python	PHP	C++	C#	C	Ruby
Kafka [2]	(ja)	ja	(ja)	(ja)	(ja)	(ja)	(ja)	(ja)
RabbitMQ [15]	(ja)	ja	(ja)	(ja)	(ja)	ja	ja	(ja)
NSQ [13]	ja	(ja)	ja	(ja)	(ja)	ja	(ja)	(ja)

Abbildung 3.1: Kompatibilitätsmatrix Message Broker. Drittanbieterbibliotheken in Klammern, soweit vom Hersteller empfohlen

Nachricht verloren geht und auch keine Nachricht mehr als einmal zugestellt wird, ist von allen Varianten die ressourcenintensivste. [5]

3.1.3 Queues

Da nicht jede Implementierung eine Queue verwendet, spielt diese nur bei Verwendung eine Rolle. Jedoch sind hier auch wieder Limitierungen von Hardware anzutreffen. So kann eine Queue nicht unendlich viele Nachrichten speichern, da meist begrenzter Speicherplatz, insbesondere bei flüchtigen Speichermedien vorliegt. Werden Nachrichten beispielsweise nach dem Versand direkt wieder gelöscht, so wird natürlich weniger Speicher benötigt. Der **Speicherort** (etwa auf einer Festplatte oder im Arbeitsspeicher), und die Option zur **Langzeitspeicherung** sind also interessant.

Da bei der Verwendung von Queues alle Nachrichten diese durchlaufen, ist außerdem relevant, ob die Reihenfolge dieser dabei erhalten wird. Man spricht in diesem Kontext von **Ordnungsgarantien**. Dabei kann eine MOM *no order*, also keine Ordnungsgarantien, *partitioned order*, also eine Ordnungsgarantie über eine Teilmenge von Nachrichten, oder *total order*, also eine absolute Ordnung, anbieten. [5]

Über die erarbeiteten Punkte hinaus ist es schwer, Eigenschaften zu finden, in denen sich Implementierungen wesentlich unterscheiden. Die meisten aktuellen Implementierungen sind auf einen bestimmten Anwendungsfall ausgerichtet, was den Vergleich außerhalb genannter Eigenschaften erschwert [11].

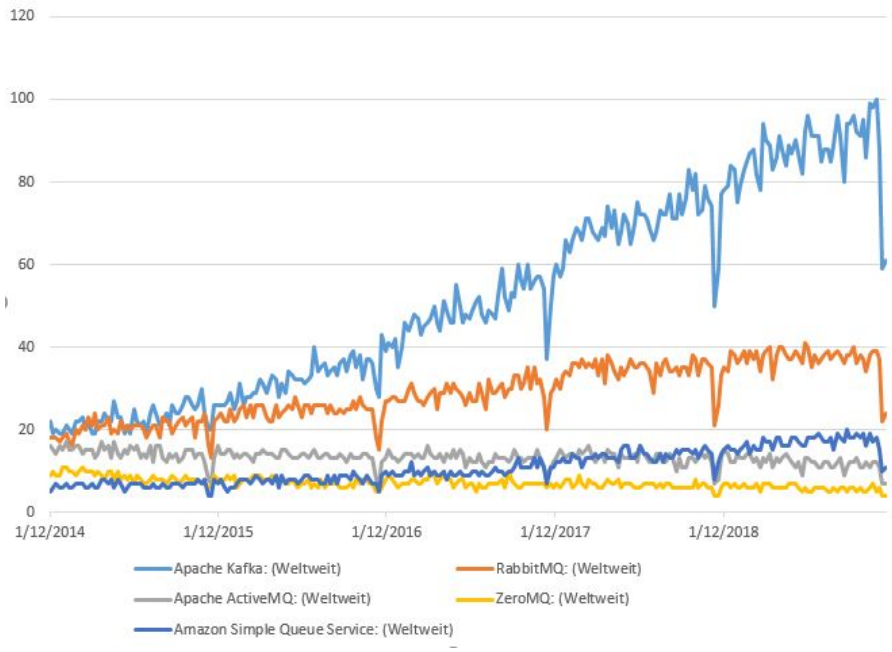


Abbildung 3.2: Relatives Suchinteresse in 5 Jahren; Quelle: Google Trends

3.2 Aktuelle Implementierungen

Für den Vergleich wurden Kafka und RabbitMQ gewählt, da sie in den letzten Jahren am meisten Aufmerksamkeit erhalten haben. Dies ist zu sehen in Darstellung 3.2, die das Volumen der jeweiligen Suchanfragen für Implementierungen auf der Google Websuche zeigt.

In den letzten Jahren gab es auf dem Gebiet viele Neuentwicklungen, wie beispielsweise NSQ. Dieses funktioniert komplett ohne zentrale Instanz als Peer-to-Peer Ansatz. NSQ wird mit den restlichen Implementierungen verglichen, um die ansatzbedingten Unterschiede aufzuzeigen und die Allgemeingültigkeit des Katalogs zu demonstrieren. Es wird statt ZeroMQ analysiert, da es aktueller ist und nach dem gleichen Prinzip funktioniert. ActiveMQ ist eine ältere Implementierung, die JMS unterstützt. Sie ist aufgrund ihrer geringen Verwendung nicht im Vergleich inbegriffen. Amazon Simple Queue Service ist ein Cloud-Dienst und daher nicht vergleichbar.

3.2.1 Apache Kafka

Apache Kafka, erstmals erschienen 2011, ist ein Open Source Projekt zur Verarbeitung von Datenströmen. Es wurde von der Firma LinkedIn entwickelt und später Teil der Apache Software Foundation. Es fokussiert sich auf sogenannte *Streams* von Daten, also die Speicherung und Verarbeitung von Datenströmen. Zwar war Messaging nicht das Hauptziel bei der Entwicklung Kafkas, allerdings findet es große Verbreitung hierfür. Kafka bietet vereinfachte Skalierbarkeit durch Clustering. Diese Cluster können sich über mehrere physische Computer erstrecken [1].

Kafka basiert auf einem simplen Topic-Routing und umfasst eine Queue, in der Messages gespeichert werden. Consumer können zusätzlich zum gewöhnlichen Publish-Subscribe Pattern die Nachrichten auch im Nachhinein noch beliebig aus der Queue abrufen. Dabei wird intern jede Nachricht an eine sogenannte *Partition* angehängt. Dies ist nichts anderes als eine Queue mit einer FIFO-Konfiguration, allerdings wird während des Clustering unter Umständen eine identische Queue mehrmals vorgehalten. Auch kann es vorkommen, dass eine Partition für einen physischen Host zu groß wird, und auf mehrere Hosts aufgeteilt wird. Auf diese Weise ist Kafka sehr gut skalierbar. Weiter umfasst Kafka nicht nur APIs für Producer und Consumer, sondern auch für Streams. Diese können von Anwendungen zum Transformieren gesamter Streams verwendet werden, beispielsweise um von einem Topic nach Verarbeitung der Daten diese direkt an ein anderes Topic weiterzugeben. [1]

Kafka ist in Scala implementiert und läuft auf der Java Virtual Machine. Somit ist eine gewisse Plattformunabhängigkeit beim Hosting gegeben. Zum Aufsetzen und Administrieren wird jedoch ZooKeeper, eine weitere Software der Apache Foundation, benötigt. Dabei kann die Provision, Replikation und Konfiguration aller Kafka-Instanzen ausschließlich über Zookeeper ablaufen. Im Vergleich zu RabbitMQ ist der Aufwand einer Installation bei weniger Instanzen hierbei höher, allerdings vereinfacht Zookeeper dafür die Administration von sehr vielen Instanzen. Der offizielle Client ist ebenfalls nur für Java/Scala verfügbar, es gibt jedoch eine Reihe an Drittanbieterbibliotheken, die von Apache empfohlen werden. Die Kompatibilität ist in Tabelle 3.1 dargestellt. Kafka unterstützt das verbreitete AMQP-Protokoll nicht. Es setzt stattdessen auf eine Eigenentwicklung. Ist die Unterstützung also eine harte Anforderung, muss auf Drittanbieterbibliotheken zurückgegriffen werden, wobei man wiederum Performanceverluste, Kompatibilitätsprobleme usw. in Kauf nehmen muss. [2, 1]

Routing Wie bereits erwähnt, basiert das Routing von Kafka auf Topics. Hierbei sind zahlreiche Konfigurationen für einzelne Topics möglich. Unter anderem kann konfiguriert werden, wie lange Nachrichten vorgehalten werden, wie viel Speicher für sie reserviert wird, ihre maximale Länge, uvm.

Die versendeten Nachrichten bestehen aus *key*, *value* und einem Zeitstempel. Sie enthalten nicht die Partition, in der sie veröffentlicht werden. Bei beispielsweise einer Partition pro Topic versendet stattdessen der Producer die Nachrichten direkt an eine Partition.

Besonderheiten Bei klassischen Publish-Subscribe Message Brokern ist es unüblich, dass Nachrichten vorgehalten werden und mehrmals abgerufen werden können. Dies wird jedoch durch die Verwendung einer Queue ermöglicht. Das kann bei der Verwendung von Kafka als MOM tatsächlich auch zu weiteren Vorteilen führen. Beispielsweise gibt es so keinen architekturbedingten Nachrichtenverlust, wenn Clients ausfallen. Diese können die Bearbeitung der Nachrichten dank der Queue zu einem späteren Zeitpunkt fortsetzen.

Des Weiteren besitzt Kafka APIs, die gezielt Import und Exporte zu Big-Data Speichersystemen unterstützen. Davon profitieren Systeme, die sehr große Datenmengen verarbeiten und kommunizieren müssen, wie im Szenario von Event Sourcing.

Analyse nach Eigenschaftenkatalog Kafka ist **at least once** zuzuordnen, wenn es um Zustellungsgarantien geht. Da die Consumer Nachrichten selbstständig und auch mehrfach abrufen können, ist davon auszugehen, dass Kafka nicht in die Gruppen *at most once* oder *exactly once* eingeordnet werden kann. [1]

Aufgrund von Kafka's Partitionen liegt eine **partitioned ordering** Garantie vor. Zwar ist in den einzelnen Partitionen eine absolute Ordnung, allerdings kann beispielsweise bei Topic-basierter Partitionierung eine spätere Nachricht in einem selten gebrauchten Topic weiter vorne stehen als eine frühere in einem größeren Topic. Durch die Partitionen wird eine Replikation von Kafka vereinfacht. Hinzu kommt eine einfache Skalierbarkeit mit Zookeeper. Im Detail wird pro Cluster, also einer Menge von zusammenarbeitenden Kafka-Instanzen, manuell ein Anführer bestimmt, der dann alle Entscheidungen über Verteilung von Replikationen von Partitionen über die Instanzen trifft. Dies bewirkt, dass bei Ausfall einer Instanz immer noch ein Replika übernehmen kann. [1]

Gerade im Bereich Durchsatz kann Kafka jedoch besonders glänzen. Da die Queue und Routing simpel gehalten sind, werden sehr hohe Durchsätze erreicht. Dabei wirkt sich die Anzahl an Partitionen indirekt proportional auf den Durchsatz aus. Es gibt mehrere Benchmarks, die die Performance von Kafka belegen. Grundsätzlich sind Schreibraten von ungefähr 2.000.000 Nachrichten pro Sekunde auf durchschnittlicher Hardware möglich. Allerdings ist die Latenz von Kafka nicht stabil [3, 5].

Zuordnung zu Use Cases Aufgrund der beschriebenen architekturellen Gegebenheiten Kafkas ist dieses für einige Use Cases besonders geeignet, für andere dafür eher weniger. Beispielsweise sollte das System keine hohen Anforderungen an das Routing mit sich bringen, da dies bei Kafka überaus simpel gehalten ist. Allerdings ergeben sich dadurch immense Performancegewinne, von denen beispielsweise Microservices oder Event Sourcing Szenarien sehr profitieren können. Des Weiteren können bei Kafka Nachrichten nachträglich erneut abgerufen werden, was es stark von anderen Implementierungen abhebt. Oftmals wird Kafka auch wegen seiner hohen Performance gewählt. Diese liegt im Bereich des fünffachen von RabbitMQ [5]. Da keine AMQP-Unterstützung vorhanden ist, wird man jedoch im Enterprise Umfeld selten Kafka vorfinden.

3.2.2 RabbitMQ

RabbitMQ ist eine von Pivotal Software entwickelte Message Oriented Middleware, welche das AMQP Protokoll implementiert. Der Server ist in Erlang geschrieben und plattformunabhängig. Er unterstützt die in 3.1 dargestellten Programmiersprachen. Nicht nur AMQP, sondern auch eine Reihe weiterer Protokolle, wie etwa das im IoT-Umfeld beliebte MQTT, werden unterstützt [5]. Daher eignet sich RabbitMQ besonders in Bereichen, in denen mehrere Protokolle zum Einsatz kommen, oder AMQP eine harte Anforderung ist.

RabbitMQ verwendet eine Queue. Im Detail werden Nachrichten vom Producer an eine sogenannte Exchange, welche als Router für Nachrichten fungiert, gesendet. Diese Exchange entscheidet, an welche Queue die Nachricht weitergeleitet wird. Dabei basiert die Entscheidung auf vorhandenen Routingregeln. Danach wird die Nachricht in der Queue solange zwischengespeichert, bis sie zugestellt werden kann. Anders als bei RabbitMQ werden bereits zugestellte Nachrichten daraufhin gelöscht. [14]

Routing Da RabbitMQ AMQP implementiert, können sehr komplexe Routing-Logiken umgesetzt werden. Dabei ist hauptsächlich Topic-basiertes Routing relevant, da dies leichter mit dem Routing von Kafka verglichen werden kann. Hierbei wird ein sogenanntes Multipart-Routing unterstützt: Die Nachrichten werden mit Topics in der Form `a.b.c` versehen, welche dann mithilfe von Wildcards gefiltert werden können. Die Konfigurationsmöglichkeiten sind hierbei sehr vielfältig. Ferner existiert auch ein inhaltsbasiertes Routing. [14]

Besonderheiten RabbitMQ ist von Haus aus mit sehr umfangreichen Monitoring und Management Tools ausgestattet. Diese erlauben, die meisten wichtigen Aspekte auf einen Blick zu kontrollieren. In Darstellung 3.3 sieht man die Weboberfläche einer RabbitMQ Instanz, die aktiv verwendet wird. Dabei sind Statistiken zur Gesamtlast sichtbar. Ferner kann RabbitMQ bei Bedarf komplett in-memory, also ohne Nachrichten auf einer Festplatte zwischenzuspeichern, sondern nur im Arbeitsspeicher, betrieben werden.

Die Consumer werden von RabbitMQ aktiv protokolliert. So besteht stets Transparenz, welcher Consumer welche Nachrichten bereits empfangen hat. Außerdem können Nachrichten pro Queue mit einer Time to Live, also einer Gesamt-

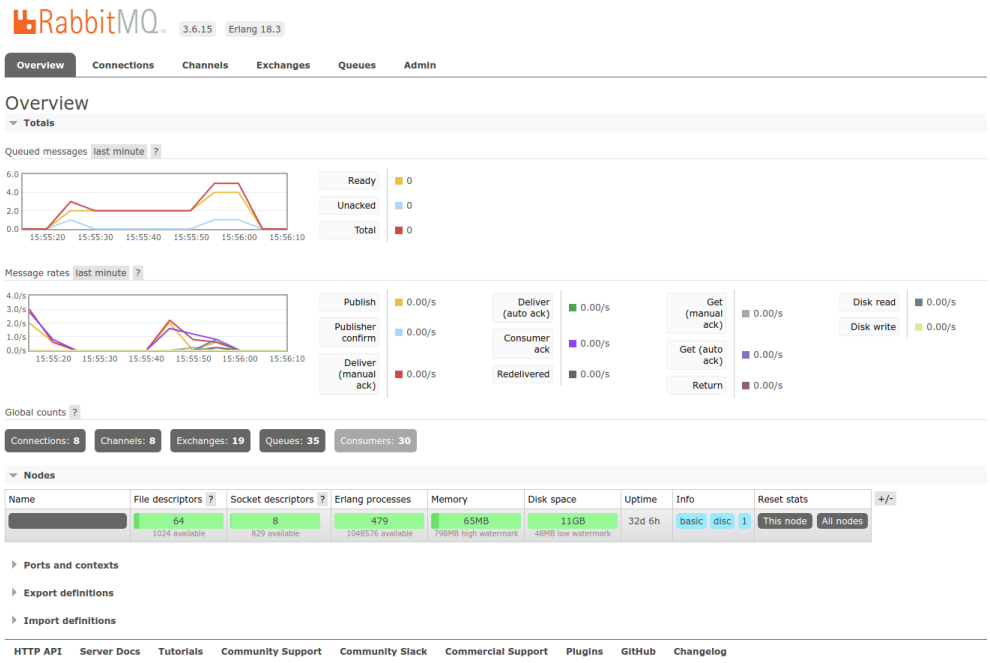


Abbildung 3.3: Weboberfläche von RabbitMQ

dauer, die eine Nachricht im System existieren darf, versehen werden. Dies macht vor allem im Kontext von Echtzeit-Daten Sinn.

Auch unterstützt RabbitMQ direkte Antworten auf Nachrichten. Bei Bedarf kann auf eine Nachricht direkt und ohne entsprechende Antwortqueues geantwortet werden.

Analyse nach Eigenschaftenkatalog Die von RabbitMQ übernommenen Garantien bezüglich der Zustellung von Nachrichten sind, wie auch bei Kafka, **at least once**. Es besteht also Risiko, dass Nachrichten doppelt gesendet werden, jedoch nicht, dass sie nie gesendet werden.

Anders verhält es sich bei der Sortierung. RabbitMQ unterstützt **global order**. Die Exchange übergibt einer Queue stets die Nachrichten in Batches, die vorsortiert sind. Verliert ein Client also eine Message aus einer Batch, stellt dies RabbitMQ fest, und kann die einzelne Nachricht erneut senden. Das Routing kann, wie bereits erwähnt, sehr komplex sein, und es werden neben Topic basiertem Routing noch weitere Arten unterstützt.

RabbitMQ unterstützt High Availability durch Replikation vollständiger Exchanges. Allerdings ist der damit verbundene Konfigurationsaufwand im Vergleich zu Kafka erheblich. Ferner kann relativ einfach skaliert werden, indem dem Cluster neue Instanzen zugefügt werden. Dabei können neue Queues auf der neuen Instanz zum Master werden, das heißt, die andere Instanz kopiert sie zwar, aber Consumer verwenden bevorzugt diesen Master. Anders als bei Kafka kann jedoch keine bereits bestehende Queue eine neue Instanz zu Ihrem Master machen. Somit ist die Skalierung mit mehr Aufwand als bei Kafka verbunden.

Use Cases Aufgrund der beschriebenen Besonderheiten RabbitMQs eignet sich dieses hervorragend für Anwendungen, die keine extremen Datenmengen verarbeiten müssen. Dank des sehr fortgeschrittenen Routings von AMQP können leicht Anwendungen mit sehr vielen Clients bedient werden.

Ein idealer Use Case ist beispielsweise IoT. Sowohl im privaten, wie im industriellen Umfeld, kann von geringen Latenzen und dem Support des MQTT Protokolls profitiert werden. Des Weiteren trägt ein fortgeschrittenes Monitoring zum einfachen Debuggen von Anwendungen bei. Auch wird RabbitMQ häufig im Enterprise Bereich eingesetzt. Durch die Unterstützung von mehreren standardisierten Protokollen wird eine Integration in ein Umfeld mit stark heterogenen Teilnehmern und auch die Anbindung von älteren Systemen an RabbitMQ begünstigt.

3.2.3 NSQ

NSQ ist eine relativ neue Implementierung des Unternehmens bitly, die ähnliche Möglichkeiten wie eine MOM bietet. Dabei entfällt jedoch eine zentrale Instanz. Das bedeutet, es gibt keinen Server, mit dem alle Kommunikationspartner verbunden sind, sondern alle Teilnehmer sind untereinander verbunden. Damit man diese Verbindungen nicht verwalten muss, gibt es eine *Service Discovery*. Diese kann andere Instanzen entdecken und stellt so ein Netzwerk her. Dezentrale Lösungen wie diese existierten jedoch schon vor NSQ, beispielsweise ZeroMQ. Der Client sowie die Service Discovery von NSQ sind in Go implementiert. [13]

Besonderheiten Wie bereits erwähnt, entfällt eine zentrale Instanz, über die die Nachrichten geleitet werden. Stattdessen startet jeder Teilnehmer eine Service Discovery, verbindet sich mit dem Netzwerk und kann anschließend Kanälen beitreten oder auf Topics abonnieren. Wie bereits bei den anderen Lösungen werden Nachrichten nach dem Publish-Subscribe Pattern durch das Netzwerk propagiert. Dementsprechend wird eine Queue lokal bei jedem Teilnehmer gehalten. Kann eine Nachricht nicht direkt in das Netzwerk eingespeist werden, wird sie hier gespeichert. NSQ kann zusätzlich zu den unterstützten Clientbibliotheken (vermerkt in 3.1) eine HTTP-API anbieten, wodurch es auch in anderen Sprachen genutzt werden kann. Des Weiteren bietet es eine einfache Integration eines Monitoring sowie eine Weboberfläche zur Administration an. [13]

Analyse NSQ unterstützt kein standardisiertes Protokoll, es setzt auf eine Eigenlösung. Architekturbedingt werden Nachrichten ohne jegliche Ordnungsgarantie, dafür aber **at least once** übertragen. Eine Skalierbarkeit durch Replikation entfällt, da für jeden Teilnehmer eine eigene Instanz gestartet wird. Dank der minimalen Architektur wird ein hoher Durchsatz und eine gleichbleibend niedrige Latenz erreicht. Die Queue kann wahlweise auf Festplatte oder Arbeitsspeicher gehalten werden, eine Langzeitspeicherung entfällt. [13]

NSQ eignet sich somit nicht für Event Sourcing, da eine Langzeitspeicherung erforderlich ist. Weil entsprechende Protokolle nicht unterstützt werden, ist es auch für den IoT-Bereich ungeeignet. Dank hohem Durchsatz und geringer Latenz bietet sich NSQ für den Einsatz mit Microservices an.

3.3 Vergleich

Wie sich in den letzten Abschnitten gezeigt hat, gibt es einige große Unterschiede zwischen RabbitMQ und Kafka. Während RabbitMQ sehr feine Regelungen beim Routing anbietet, spricht die Performance für Kafka. Allerdings sind die Protokolle nicht standardisiert und RabbitMQ unterstützt nicht nur AMQP, sondern gleich mehrere Protokolle. Kafka bietet wiederum eine langfristige Speicherung für Nachrichten.

Während Kafka im Bereich Microservices oder Event Sourcing seine Stärken ausspielen kann, überzeugt RabbitMQ bei IoT und im Enterprise Bereich. Doch trotz aller Unterschiede bieten sowohl RabbitMQ als auch Kafka nach wie vor einen Message Broker an. Dazu im Kontrast steht NSQ repräsentativ für viele neue Lösungen, die auf eine zentrale Instanz verzichten. Architekturbedingt kann NSQ allerdings keine Ordnungsgarantien oder Langzeitspeicherung von Nachrichten bieten. [12, 3]

Allerdings gibt es einen Weg, Vorteile Kafkas und RabbitMQs kombiniert zu nutzen. Beispielsweise kann man RabbitMQ verwenden, und Kafka einzelne Topics speichern lassen. Hierbei profitiert man von den regulären Eigenschaften RabbitMQs, allerdings erhält man gleichzeitig bei Bedarf eine verteilte Langzeitspeicherung für wichtige Daten. Dieses Szenario ist also vorteilhaft, wenn man zwar RabbitMQ verwenden würde, allerdings eine Langzeitspeicherung zumindest für einzelne Topics benötigt wird [5].

In einem anderen Beispiel könnte man Kafka verwenden, um Nachrichten entgegenzunehmen, und eine RabbitMQ Instanz pro Topic, um diese gezielter zu routen. Hierbei profitiert man von dem hohen Durchsatz Kafkas, wobei man gleichzeitig das fortgeschrittene Routing RabbitMQs verwenden kann. Dabei gehen Nachrichten nicht verloren, da Kafka sie speichert. Der Aufbau ist also geeignet, wenn man das Routing RabbitMQs benötigt, aber die Performance RabbitMQs zu gering ist [5].

Zusammenfassend können in Tabelle 3.2 noch einmal die wichtigsten Vergleichspunkte abgelesen werden. Dargestellt ist der erarbeitete Eigenschaftenkatalog, ausgefüllt für Kafka, RabbitMQ, und NSQ.

Eigenschaft	**RabbitMQ**	**Apache Kafka**	**NSQ**
Standardisierte Protokolle	AMQP, MQPP	-	-
Clientbibliotheken	alle	alle	alle
Möglichkeit zur Skalierung	Replikation	Replikation	-
Durchsatz	hoch	sehr hoch	sehr hoch
Latenz	stabil	instabil	stabil
Routingoptionen	Topic, Channel, Content	Topic	Topic, Channel
Zustellgarantien	at least once	at least once	at least once
Speicherort	RAM	Festplatte	RAM oder Festplatte
Langzeitspeicherung	nicht unterstützt	unterstützt	nicht unterstützt
Ordnungsgarantien	partitioned order	partitioned order	no order

Tabelle 3.2: Eigenschaftenkatalog Kafka, RabbitMQ, NSQ

4

Fallstudie

4.1 Ausgangssituation

Im Rahmen der Arbeit soll wie bereits erwähnt der Eigenschaftenkatalog mithilfe eines Fallbeispiels evaluiert werden. Hierbei ist dieses Szenario durch den Firmenbetreuer CHECK24 Versicherungsservice GmbH gegeben.

Diese ist für das *Versicherungscenter*, eine Webanwendung zum Verwalten und Vergleichen von Versicherungen und Dokumenten, zuständig. Kunden können hier abgeschlossene Verträge inklusive aller damit verbundenen Dokumente und Informationen einsehen und sogar selbst Fremdverträge hinzufügen, um diese dort zu verwalten. Dabei durchläuft ein Vertrag, sobald er online abgeschlossen oder hinzugefügt wurde, eine Reihe von Zustandsänderungen. Es können beispielsweise von Versicherungen Daten angefordert werden, um vorhandene Daten zu ergänzen, oder Kunden können neue Angaben machen. Auch bei Vertragsverlängerungen oder Kündigungen ändern sich Daten. Hierbei wird diese Arbeit vollständig von Microservices erledigt, und basiert teilweise auf Daten und APIs anderer Teams.

Die Microservices laufen hierbei in jeweils ihrem eigenen Dockercontainer. Ein Dockercontainer ist eine vom Betriebssystem getrennte, virtualisierte Umgebung, in der eine Anwendung inklusive aller Abhängigkeiten betrieben werden kann [6]. Zudem werden für jeden Microservice zwei Instanzen betrieben, und jede Anfrage durch ein Loadbalancing geroutet. Dabei wird zusätzlich auf Seite der Microservices verhindert, dass zweimal der gleiche Request verarbeitet wird.

Nun möchten die Entwickler und vor allem auch andere Teams in Zukunft nachverfolgen können, wann welche Änderungen gemacht wurden. Momentan ist dies nicht möglich, da Verträge ein einfaches Datenbankmodell sind. Zwar gibt es Datumsfelder beispielsweise für ein Kündigungsdatum, allerdings sind viele Änderungen im Nachhinein nicht nachvollziehbar. Es wäre nicht nur übersichtlicher für die Versicherungen, Kundenberater und anderen Teams, wenn man die einzelnen Änderungen nachvollziehen könnte, es würde auch das Debugging wesentlich vereinfachen, da man feststellen könnte, an welcher Stelle ein Fehler entstanden ist.

Daher wird das Vertragsmodell auf Event Sourcing umgestellt werden. Hierbei wird, wie bereits in Abschnitt 2.3 erläutert, jede Änderung als Event-Objekt mit zugehörigem Datum gespeichert. Danach ist eine Rekonstruktion möglich, da die Änderungen in der richtigen Reihenfolge hintereinander den aktuellen

Zustand ergeben [11]. Die Verwendung einer Message Oriented Middleware ist in diesem Fall ein offensichtlicher Schritt, da viele einzelne Anwendungen immer wieder Änderungen beitragen, und alle anderen Anwendungen diese zeitnah erhalten sollen. Daher muss eine konkrete Implementierung gewählt und integriert werden. Mithilfe der zuvor aufgestellten Modelle werden nun die Anforderungen erarbeitet und eine Implementierung empfohlen.

4.2 Anforderungsanalyse

Die beschriebenen Szenarien, Microservices und Event Sourcing, geben schon einen ersten Hinweis auf eine geeignete Implementierung. Um allerdings den vorhin aufgestellten Eigenschaftenkatalog optimal zu verwenden, sollten zunächst die Anforderungen im Detail analysiert werden.

In Bezug auf Zustellgarantien sind folgende Beobachtungen zu machen. Da teilweise wichtige Kundendaten bzw. Änderungen daran kommuniziert werden, darf auf keinen Fall eine Nachricht verloren gehen. *At least once* scheint angebracht zu sein. Eine Wiederholung von Nachrichten sollte jedoch eingeplant werden und wird zu zusätzlichem Implementierungsaufwand führen. Da jedoch die Events üblicherweise mit einem Timestamp versehen sein werden, sollte ein Filter bei den Clients trivial sein, da davon ausgegangen werden kann, dass keine zwei Änderungen gleichen Inhalts in der gleichen Millisekunde stattfinden.

Da Events im Nachhinein abgerufen werden müssen, um den aktuellen Zustand zu berechnen, muss die Implementierung eine Queue mit permanenter Speichermöglichkeit anbieten. Eine Ordnung der Nachrichten ist in diesem Szenario keine Primäranforderung. Zwar wäre es schlecht, wenn eine Nachricht über eine Änderung einer Versicherung vor der Nachricht über ihre Erstellung ankommt, allerdings wird keine *total order* benötigt. Da bei der Erstellung der Events ein Zeitstempel abgespeichert wird, kann die korrekte Abfolge von Events auch im Nachhinein noch rekonstruiert werden.

Da das Versicherungscenter ein wachsender Geschäftszweig der CHECK24-Gruppe ist, ist Skalierung eine große Herausforderung. Eine einfache Lösung, die auch Mittel für Hochverfügbarkeit mit sich bringt, ist gewünscht. Ebenso ist in naher Zukunft mit hohen Lasten zu rechnen. Eine MOM kann hierbei unabhängig vom restlichen System skaliert werden, da die Microservices eine Mehrfachbearbeitung von Nachrichten verhindern. So können beliebig viele Instanzen eines Services die gleiche Nachricht empfangen, ohne dass es zu Problemen kommt. So wird für eine MOM eine unabhängige Skalierung ermöglicht.

Ein fortgeschrittenes Routing wird nicht benötigt. Da die Kommunikation bisher über ein Datenbankmodell bzw. über HTTP-Calls erfolgte, werden keine komplexen Routen verwendet. Das primäre Ziel ist, die momentane Lösung durch eine zukunftstauglichere zu ersetzen, nicht den Funktionsumfang zu erweitern. Da außerdem die Middleware nur von einem Team und momentan ca.

100 Microservices verwendet wird, ist das Routing trivial.

Eine stabile Latenz ist teilweise wichtig. Für einen Kunden macht es keinen großen Unterschied, ob er die Vertragsdokumente, die eine Versicherungsgesellschaft bereitgestellt hat, einige Millisekunden später sieht. Ebenso verhält es sich bei der Kündigung oder der Abschlussbestätigung seitens der Versicherung. Es verhält sich allerdings anders, sobald der Kunde seine Verträge betrachtet. Wird der aktuelle Vertragszustand aus allen Events berechnet, und dauert dies sehr lange, so muss der Kunde auch länger darauf warten. Daher sollte in dieser Hinsicht eine stabile Latenz gewährleistet sein.

Hinsichtlich einer Queue ist eine langfristige Speichermöglichkeit wünschenswert. Eventuell können zu einem späteren Zeitpunkt Caching-Lösungen implementiert werden, oder der Verlauf ist für Verträge eines gewissen Alters nicht mehr relevant. Dies kann allerdings erst in der Zukunft abgewogen werden. Bis dahin sollten keinesfalls Verträge gelöscht werden, die noch nicht permanent in einer anderen Speicherlösung abgelegt wurden. Da für solche Datenmengen die Speicherung im Arbeitsspeicher sehr teuer werden kann und die Geschwindigkeit ausreicht, genügt die Speicherung auf Festplatte.

Somit sind die in Abschnitt 3.1 erarbeiteten Kriterien betrachtet und nach ihrer Wichtigkeit beurteilt worden. Dieses Ergebnis ist nochmals in Tabelle 4.1 festgehalten.

Eigenschaft	**Anforderung?**
Standardisierte Protokolle	nein
Clientbibliotheken	Javascript
Möglichkeit zur Skalierung	unabhängig
Hoher Durchsatz	ja
Stabile Latenz	teilweise ja
Routing	Topic ausreichend
Zustellgarantien	*at least once* ausreichend
Speicherort	Festplatte bevorzugt
Langzeitspeicherung	ja
Ordnungsgarantien	partitioned

Tabelle 4.1: Anforderungstabelle CHECK24 Versicherungsservice GmbH

4.3 Empfehlung

Aufgrund der im vorhergehenden Abschnitt aufgestellten Anforderungen kann nun eine Empfehlung ausgesprochen werden. Die Punkte deuten fast ausschließlich auf Apache Kafka hin. Es eignet sich sehr gut, um eine Microservice-Infrastruktur wie die oben beschriebene zu betreiben. Zudem eignet es sich dank der Langzeitspeicherung für Event Sourcing. Werden Änderungen an Verträgen gemacht, können diese als Message in Kafka abgelegt werden, und Worker können diese dann stückweise abarbeiten, bzw. andere Anwendungen diese abrufen und sind damit immer auf dem neuesten Stand. Somit eignet sich in diesem Szenario Kafka besser als RabbitMQ. Eine Erweiterung um die Routing-Kapazitäten RabbitMQs ist vorerst nicht notwendig. Es entfällt also auch eine Erweiterung um das Routing von RabbitMQ.

Allerdings ergeben sich mit Kafka auch Nachteile. Beispielsweise ist wie bereits erwähnt das Hosting nicht trivial. Es werden Zookeeper bzw. eine Sammlung von Skripten verwendet, um Kafka zu administrieren. Auch würde dies eine Abweichung zum sonstigen Hosting bedeuten, da die Microservices alle in sauber getrennten, eigenen Dockercontainern betrieben werden. Dem Problem kann jedoch entgegengewirkt werden, indem man Kafka auch in einem Dockercontainer betreibt. Hierbei sollte bei der Skalierung dafür gesorgt werden, dass mindestens zwei verbundene Instanzen auf mehr als einem physischen Server laufen.

Des Weiteren werden mit Kafka keine stabilen Latenzen erreicht [3]. Dies kann zu Problemen führen, wenn ein Kunde seine Vertragsdaten im Frontend betrachtet. Hierbei werden vom zugehörigen Microservice die Daten bereitgestellt. Braucht dieser also zu lange, um alle Events für einen Vertrag abzurufen, muss der Kunde länger warten.

Dieses Problem kann auf zwei Arten gelöst werden. Beispielsweise kann der aktuelle Zustand in einer Datenbank mitgeführt werden. Bei einem Änderungsevent muss so nur ein simpler Microservice diesen Eintrag des jeweils aktuellen Zustands aktualisieren. Auf diese Weise können Microservices, die den aktuellen Zustand eines Vertrags benötigen, diesen sehr schnell abrufen. Der Aufwand hierfür ist gering, besagte Microservices können weiterhin aus der bereits existierenden Datenbank abfragen, und ein simpler Microservice, der bei Change Events diese updated, ist dank der MOM schnell integriert. Somit kann trotz instabiler Latenzen Kafka verwendet werden.

Die zweite Lösung wäre, RabbitMQ aufgrund seiner geringen Latenzen zu verwenden, und die Speicherung der Events in einer externen Datenbank vorzunehmen. Hierbei würde man von der erleichterten Administration von RabbitMQ profitieren, und gleichzeitig eine ausreichend schnelle Langzeitspeicherung selbst hinzufügen. Allerdings wäre der Gesamtaufwand wesentlich höher, als der Aufwand der vorherigen Lösung. Dies würde also nur in Frage kommen, wenn eine langfristige Speicherung von Events in Kafka ungeeignet ist und der Mehraufwand in Kauf genommen werden kann.

Die Empfehlung fällt auf Kafka. Hiermit können mit geringem Aufwand die Anforderungen befriedigt werden. Unzureichend erfüllte Anforderungen können ebenfalls einfach erfüllt werden. Andere Lösungen sind möglich, stellen jedoch einen erheblichen Mehraufwand dar.

5

Schluss

5.1 Fazit

In dieser Arbeit wurden die Grundlagen von Message Oriented Middlewares beleuchtet. Dabei wurden der allgemeine Aufbau und die Funktion einzelner Komponenten erklärt. Vor- und Nachteile einer MOM wurden abgewägt. Es wurden außerdem einige praxisnahe Szenarien, in denen oft eine Message Oriented Middleware Einsatz findet, aufgezeigt. Hierbei wurde erklärt, warum sie jeweils von dieser Verwendung profitieren.

Davon ausgehend wurde ein Eigenschaftenkatalog erarbeitet, mithilfe dessen eine erleichterte Analyse verschiedener Implementierungen ermöglicht wurde. Dieser umfasst die in der Praxis wichtigen Kriterien, in denen sich Implementierungen unterscheiden können. Dabei wurde jede dieser Eigenschaften verständlich begründet, sodass selbst fachfremde sie verwenden können, um eine Auswahl zu treffen.

Anschließend wurden die verbreiteten Implementierungen analysiert und miteinander auf Grundlage dieses Eigenschaftenkatalogs verglichen. Zusätzlich wurde eine grundsätzlich verschiedene Neuentwicklung analysiert. Hierdurch wurde gezeigt, dass der Katalog anwendbar ist und auch für zukünftige Implementierungen genutzt werden kann.

Zur Validierung des Eigenschaftenkatalogs wurde dieser abschließend auf ein praktisches Szenario angewendet. Eine Integration in den durch die CHECK24 Versicherungsservice GmbH gegebenen Anwendungsfall wurde geplant und in allen Punkten des Katalogs untersucht. Anschließend wurde eine Empfehlung ausgesprochen. Diese wurde auch vom Rest des Teams nach gemeinsamer Analyse getragen.

Die Arbeit informiert ausführlich über die führenden Implementierungen, gibt jedoch gleichzeitig ein Werkzeug mit an die Hand, mit dem Neuentwicklungen eingeordnet werden können. Wie demonstriert wurde, kann sie als Grundlage für Entscheidungen in echten Anwendungsfällen dienen.

5.2 Ausblick

Eine Message Oriented Middleware findet in vielen modernen Systemen Anwendung. Eine sauber getrennte, asynchrone, und vor allem performante Kommunikationslösung für Softwaresysteme wird auch in nächster Zukunft aus den meisten Systemen nicht wegzudenken sein. Dabei wächst das Angebot an Lösungen stetig. Die Spezialisierung der einzelnen Lösungen ist dabei hoch, die meisten Neuentwicklungen sind auf einen bestimmten Anwendungsfall zugeschnitten. Somit wäre es sicher interessant, den Vergleich dieser Arbeit auf weitere Implementierungen auszuweiten. Diese werden sich jedoch untereinander weniger stark unterscheiden als Kafka und RabbitMQ.

Auch die neueste Entwicklung - Brokerlose MOMs - könnten mit den erarbeiteten Metriken leicht eingeschätzt werden, auch wenn ihre detaillierte Behandlung den Rahmen der Arbeit gesprengt hätte. Diese Entwicklung lohnt es sich jedoch in jedem Fall zu verfolgen. Dabei gibt es andere Mechanismen zu erforschen und optimieren, wie beispielsweise die zwingend notwendige Service Discovery, mit der solche Peer to Peer Systeme funktionieren. Da diese Entwicklungen noch relativ neu sind, wird hier in den nächsten Jahren noch viel weiterentwickelt werden.

RabbitMQ sticht mit seinen Enterprise Eigenschaften hervor, aufgrund derer es trotz vergleichsweise niedrigem Durchsatz immer noch große Verbreitung findet. Ob RabbitMQ in naher Zukunft von einer anderen Implementierung abgelöst werden kann, die einen ähnlichen Support für viele Protokolle anbietet, bleibt spannend. Ebenso wäre eine Entwicklung interessant, die neue Architekturen, wie Peer to Peer MOMs, in Verbindung mit dem Support für standardisierte Protokolle bringt. Dies wäre beispielsweise für den IoT-Bereich relevant.

Es lässt sich abschließend sagen, dass in den nächsten Jahren sicher noch viel auf dem Gebiet der Message Oriented Middleware passieren wird. Diese Arbeit hat eine Übersicht über den aktuellen Stand der Entwicklung gegeben, während die verwendeten Methoden auch in der Zukunft verwendet werden können.

Anhang

Literatur

[1] *Apache Kafka*. https://kafka.apache.org/. (Accessed on 11/15/2018).

[2] *Apache Kafka Clients*. https://cwiki.apache.org/confluence/display/KAFKA/Clients/. (Accessed on 11/15/2018).

[3] LinkedIn Engineering Blog. *Benchmarking Apache Kafka*. https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines. (Accessed on 11/15/2018).

[4] Edward Curry. „Message-oriented middleware“. In: *Middleware for communications* (2004), S. 1–28.

[5] Philippe Dobbelaere und Kyumars Sheykh Esmaili. „Kafka versus RabbitMQ: A comparative study of two industry reference publish/subscribe implementations: Industry Paper“. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. ACM. 2017, S. 227–238.

[6] Docker. *What is a Container*. https://www.docker.com/resources/what-container. (Accessed on 01/28/2019).

[7] Patrick Th Eugster u. a. „The many faces of publish/subscribe“. In: *ACM computing surveys (CSUR)* 35.2 (2003), S. 114–131.

[8] Martin Fowler. *Event sourcing*. https://martinfowler.com/eaaDev/EventSourcing.html. 2005.

[9] Martin Fowler. *Microservices*. https://martinfowler.com/articles/microservices.html. 2014.

[10] *Github Octoverse*. https://octoverse.github.com/projects. (Accessed on 12/18/2018).

[11] Martin Kleppmann. *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media, Inc., 2017.

[12] Simon MacMullen. *RabbitMQ Performance Measurments*. http://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2/. (Accessed on 11/15/2018). Apr. 2012.

[13] *NSQ Documentation*. https://nsq.io/. (Accessed on 12/13/2018).

[14] *RabbitMQ Documentation*. https://www.rabbitmq.com/documentation.html. (Accessed on 11/15/2018).

[15] *RabbitMQ Documentation*. https://www.rabbitmq.com/devtools.html. (Accessed on 11/15/2018).

[16] Mohammad Abdur Razzaque u. a. „Middleware for Internet of Things: A survey." In: *IEEE Internet of Things Journal* 3.1 (2016), S. 70–95.

[17] Matthew Sackman. *Sizing your Rabbits*. https://www.rabbitmq.com/blog/2011/09/24/sizing-your-rabbits/. (Accessed on 11/15/2018). Sep. 2011.

[18] Andrew S Tanenbaum und Maarten Van Steen. *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[19] Steve Vinoski. „Advanced message queuing protocol". In: *IEEE Internet Computing* 10.6 (2006).

Abbildungsverzeichnis

Tabellenverzeichnis