HTWD

**Masterarbeit**

# Evaluierung von zukunftsfähigen Middleware-Lösungen für die hochautomatisierte Halbleiterfertigung


| | |
|---|---|
| vorgelegt von: | Lucas Dachwitz |
| Studienbereich: | Angewandte Informatik |
| Ort: | HTW Dresden |
| Matrikelnummer: | 55283 |
| Erstgutachter: | Prof. Dr.-Ing. Jörg Vogt |
| Zweitgutachter: | M.Eng. Tarek Daood |


| | |
|---|---|
| Abgabedatum: | 01.09.2025 |

# Abstract

Diese Masterarbeit befasst sich mit der Evaluierung zukunftsfähiger Middleware-Lösungen für die hochautomatisierte Halbleiterfertigung. Ausgangspunkt ist das bestehende, TIBCO-basierte Nachrichtensystem, das aufgrund hoher Kosten, eingeschränkter Skalierbarkeit und technischer Limitierungen abgelöst werden soll. Ziel der Arbeit ist es, eine Message Oriented Middleware (MOM) zu identifizieren, die den aktuellen und zukünftigen Anforderungen der X-FAB Dresden GmbH gerecht wird.

Hierzu wurden unterschiedliche MOM-Technologien (u. a. ActiveMQ Artemis, NATS und MQTT 5) untersucht und anhand definierter Kriterien bewertet. Neben theoretischen Vergleichen zu Architektur, Protokollunterstützung und Funktionsumfang wurden praxisnahe Tests durchgeführt, bei denen reale Produktionsnachrichten mit Hilfe eines von der SYSTEMA GmbH bereitgestellten Client-Server-Frameworks reproduziert und ausgewertet wurden. Diese Tests umfassten sowohl Dauerlastszenarien als auch Belastungsspitzen, um die Leistungsfähigkeit, Stabilität und Ausfallsicherheit der Systeme zu überprüfen.

Die Ergebnisse zeigen, dass insbesondere ActiveMQ Artemis die geforderten Leistungs- und Stabilitätskriterien erfüllt und aufgrund seiner modernen Architektur, Protokollvielfalt und Skalierbarkeit eine geeignete Nachfolgelösung darstellt. Darüber hinaus bietet MQTT 5 wichtige Erweiterungen für IoT-Anwendungen und vereinfacht die abteilungsübergreifende Nutzung des Systems. Abschließend wird empfohlen, Artemis als neuen Message Bus einzuführen, Langzeittests unter Produktionsbedingungen durchzuführen und die Migration schrittweise umzusetzen, um einen stabilen und nachhaltigen Übergang sicherzustellen.

# Inhaltsverzeichnis

# Abkürzungsverzeichnis

**AMQP** Advanced Message Queuing Protocol

**IoT** Internet der Dinge

**JCA** Java Connector Architecture

**JMS** Java Message Service

**JSON** JavaScript Object Notation

**MOM** Message-Oriented Middleware

**MQ** Message Queue

**MQTT** MQ Telemetry Transport

**M2M** Machine-to-Machine

**RPC** Remote Procedure Calls

**STOMP** Simple Text Oriented Messaging Protocol

**XML** Extensible Markup Language

**YAML** YAML Ain't Markup Language

# Abbildungsverzeichnis

# Listings

# Tabellenverzeichnis

# 1 Einleitung

## 1.1 Unternehmen

Die X-FAB Dresden GmbH & Co. KG ist seit 2007 ein integraler Bestandteil der international agierenden X-FAB-Gruppe. Als einer von sechs Fertigungsstandorten des Unternehmens spezialisiert sich das Werk in Dresden auf die Herstellung von 8-Zoll-Wafern mit einem besonderen Fokus auf CMOS- und MEMS-Technologien. Dabei profitiert der Standort von der umfassenden Expertise und dem technologischen Know-how der gesamten X-FAB-Gruppe, die als einer der weltweit führenden Halbleiterhersteller für spezialisierte Anwendungen bekannt ist.
Mit rund 460 engagierten Mitarbeitenden trägt X-FAB Dresden maßgeblich zur Entwicklung und Produktion innovativer Halbleiterlösungen bei. Die enge Zusammenarbeit zwischen den Standorten der Gruppe ermöglicht eine effiziente Nutzung von Synergien in den Bereichen Forschung, Entwicklung und Fertigung. Durch kontinuierliche Investitionen in modernste Produktionsverfahren und die stetige Weiterentwicklung der Technologien stärkt X-FAB Dresden seine Position als zuverlässiger Partner für industrielle, medizinische und automobile Anwendungen. [36]

## 1.2 Ausgangslage

Die Fertigungsprozesse bei X-FAB Dresden basieren auf einer hochautomatisierten Anlagensteuerung, deren zentrale Kommunikationskomponente ein Message Bus darstellt. Der derzeit eingesetzte TIBCO Bus übernimmt hierbei die Rolle der Nachrichtenverteilung innerhalb der Produktionsumgebung, weist jedoch zunehmend kritische Einschränkungen auf. Neben hohen Betriebskosten und einer technologisch veralteten Architektur zeigen sich verstärkt technische Limitierungen, insbesondere hinsichtlich Wartbarkeit, Flexibilität und der Integration moderner Kommunikationsstandards. Diese Einschränkungen gefährden die langfristige Skalierbarkeit des Systems und stellen ein Hindernis für die Weiterentwicklung einer zukunftsfähigen, hochautomatisierten Halbleiterfertigung dar.

## 1.3 Zielstellung

Um die langfristige Wettbewerbsfähigkeit und Effizienz der Produktionsabläufe sicherzustellen, ist die Implementierung eines neuen Message Busses erforderlich. Dieser muss mindestens die bestehenden Funktionalitäten abdecken und gleichzeitig eine zukunftssichere Architektur bieten. Besondere Anforderungen an die neue Middleware umfassen die Unterstützung moderner Kommunikationsstandards wie JMS oder MQTT5 sowie eine hohe Skalierbarkeit, Stabilität und Ausfallsicherheit.

## 1.4 Lösungsweg

Da zahlreiche Message-Oriented Middleware (MOM) -Lösungen auf dem Markt existieren, ist eine umfassende Evaluierung erforderlich, um diejenige Lösung zu identifizieren, die den spezifischen Anforderungen der X-FAB Dresden am besten gerecht wird. Dabei werden neben technischen Aspekten wie Leistungsfähigkeit, Fehlertoleranz und Integration auch wirtschaftliche Faktoren berücksichtigt. Die vorliegende Masterarbeit mit dem Titel “Evaluierung von zukunftsfähigen Middleware-Lösungen für die hochautomatisierte Halbleiterfertigung” untersucht die Anforderungen und Möglichkeiten des Austauschs der bestehenden MOM innerhalb der Produktionslandschaft von X-FAB Dresden. Ziel der Arbeit ist es, eine zukunftsfähige Middleware-Lösung zu identifizieren, die den steigenden Anforderungen der Halbleiterfertigung gerecht wird und das derzeit eingesetzte System effizient ersetzen kann.

# 2 Grundlagen

## 2.1 MOM

MOM ist eine softwarebasierte Vermittlungsschicht, die eine asynchrone, zuverlässige und skalierbare Kommunikation zwischen verteilten Systemen ermöglicht. Sie dient als zentrale Kommunikationsinfrastruktur, um Nachrichten zwischen Anwendungen auszutauschen, ohne dass diese direkt miteinander verbunden sein müssen [10]. Dies reduziert die Abhängigkeiten zwischen Systemkomponenten und erhöht die Flexibilität und Skalierbarkeit in komplexen IT-Architekturen. MOM folgt dem Paradigma des Message Passing, wo Informationen in Form von Nachrichten zwischen verschiedenen Systemkomponenten übertragen werden. Diese Nachrichten können sowohl synchron als auch asynchron verarbeitet werden, wobei asynchrone Kommunikation eine höhere Entkopplung zwischen Sender- und Empfängeranwendungen ermöglicht [14]. Im Gegensatz zu traditionellen Integrationsmethoden wie Remote Procedure Calls (RPC) oder datenbankbasierter Kommunikation bietet MOM eine lose Kopplung, die Systeme weniger anfällig für Änderungen und Skalierungsanforderungen macht.

Ein zentraler Bestandteil von MOM ist die Bereitstellung von Nachrichtenwarteschlangen (Message Queue (MQ)) oder Publish-Subscribe-Mechanismen. Diese ermöglichen es, Nachrichten temporär zu speichern und sicherzustellen, dass sie auch bei kurzfristigen Ausfällen von Empfängersystemen nicht verloren gehen. Darüber hinaus sorgt MOM für eine geordnete und priorisierte Verarbeitung der Nachrichtenströme, was insbesondere in hochverfügbaren und latenzsensitiven Anwendungen von Bedeutung ist [8]. Moderne MOM-Lösungen sind darauf ausgelegt, verschiedene Kommunikationsstandards zu unterstützen. Dazu gehören:

**Java Message Service (JMS)** Eine standardisierte Messaging-API, die im Rahmen von Jakarta EE (ehemals Java EE) entwickelt wurde und die asynchrone Kommunikation zwischen Java-Anwendungen über Nachrichten unterstützt. JMS abstrahiert von der konkreten Implementierung der Messaging-Infrastruktur und bietet eine einheitliche Schnittstelle für verschiedene Nachrichtensysteme [11].
Zentral sind zwei Kommunikationsmodelle: das Point-to-Point-Modell, bei dem Nachrichten über Queues zwischen einem Sender und einem Empfänger ausgetauscht werden, sowie das Publish/Subscribe-Modell, bei dem Nachrichten über Topics an mehrere Abonnenten verteilt werden [17]. Durch diese Modelle ermöglicht JMS eine lose Kopplung von Systemkomponenten und unterstützt gleichzeitig eine hohe Skalierbarkeit und Zuverlässigkeit – typische Anforderungen in verteilten Unternehmensanwendungen [35].

**MQ Telemetry Transport (MQTT)** Ein leichtgewichtiges Protokoll, das speziell für Machine-to-Machine (M2M) -Kommunikation und das Internet der Dinge (IoT) entwickelt wurde und ressourcenschonende Übertragungen ermöglicht.
Ursprünglich von IBM entwickelt, zeichnet sich MQTT durch seine geringe Bandbreitennutzung und minimale Anforderungen an die Gerätehardware aus, was es besonders geeignet für Umgebungen mit eingeschränkten Ressourcen macht [27].

Dank dieser Flexibilität wird MOM in zahlreichen Anwendungsbereichen eingesetzt, darunter hochautomatisierte Fertigungsprozesse, Finanztransaktionen, vernetzte Fahrzeuge und Cloud-Computing-Umgebungen. Die Fähigkeit, verschiedene Systeme miteinander zu integrieren und gleichzeitig eine robuste und ausfallsichere Kommunikation zu gewährleisten, macht MOM zu einer Schlüsseltechnologie für moderne IT-Architekturen.

### 2.1.1 Weitere Kommunikationsstandards

Neben den in dieser Arbeit primär betrachteten Standards *MQTT* und *JMS* existieren weitere etablierte Protokolle die in verschiedenen Message-Oriented-Middleware-Systemen zum Einsatz kommen. Je nach technischer Ausrichtung, Anwendungsfall und Performance-Anforderung bieten diese Protokolle spezifische Vorteile oder Einschränkungen.

Nachfolgend werden zwei zusätzliche, in der Industrie weit verbreitete Standards vorgestellt: AMQP (Advanced Message Queuing Protocol) sowie STOMP (Simple Text Oriented Messaging Protocol). Beide Protokolle werden von mehreren gängigen Message-Brokern wie RabbitMQ oder ActiveMQ Artemis unterstützt und eignen sich insbesondere für textbasierte Kommunikationsszenarien.

Die Zielsetzung dieses Abschnitts ist es, ein grundlegendes Verständnis dieser beiden Standards zu vermitteln, ihre Charakteristika mit Blick auf Integration, Flexibilität und Komplexität zu analysieren und ihre potenzielle Eignung im Kontext der vorliegenden Anforderungen zu bewerten.

**Advanced Message Queuing Protocol (AMQP)** AMQP ist ein offener, standardisierter Kommunikationsstandard, der ursprünglich für den Finanzsektor entwickelt wurde und heute vor allem durch RabbitMQ bekannt ist. Er ermöglicht zuverlässige, interoperable und plattformübergreifende Messaging-Kommunikation mit garantierter Zustellung [29].

Das AMQP-Protokoll definiert sowohl die semantische Struktur der Nachrichtenübermittlung (Queues, Exchanges, Bindings) als auch deren Codierung, Routing und Transport.

Es unterstützt verschiedene Kommunikationsmodelle – darunter *Point-to-Point*, *Publish/-Subscribe* sowie *Request/Response* – und erlaubt präzise Steuerung über Delivery Guarantees (z. B. at-least-once, exactly-once).

Ein Vorteil von AMQP liegt in der detaillierten Kontrollierbarkeit der Nachrichtenflüsse und der Möglichkeit zur Integration mit bestehenden Standards (z. B. TLS, SASL für Sicherheit). Gleichzeitig kann die Flexibilität des Protokolls auch zu einem erhöhten Konfigurations- und Verwaltungsaufwand führen. Besonders in großen Systemlandschaften mit heterogenen Komponenten ist AMQP jedoch eine robuste Alternative zu proprietären Messaging-Protokollen.

**Simple Text Oriented Messaging Protocol (STOMP)** STOMP ist ein leichtgewichtiges, textbasiertes Protokoll für Message-Oriented-Middleware, das auf der Client-Server-Architektur basiert. Im Gegensatz zu binären Protokollen wie AMQP oder MQTT setzt STOMP auf eine klar strukturierte, lesbare Nachrichtenkodierung, die sich an das Format von HTTP orientiert [31].

Ein wesentliches Merkmal von STOMP ist seine Einfachheit: Nachrichten bestehen aus einem Command, Header-Feldern und einem optionalen Body. Diese Struktur ermöglicht eine intuitive Interpretation der übertragenen Inhalte, auch ohne spezialisierte Tools. Dadurch eignet sich STOMP besonders für Entwicklungs- und Debuggingzwecke sowie für Szenarien, in denen textuelle Datenstrukturen wie JSON oder XML verwendet werden – wie es im Rahmen dieser Arbeit der Fall ist.

STOMP ist kein vollständiges Messaging-Framework, sondern agiert als einheitliche Protokollschicht zwischen Clients und einem Message-Broker. Es wird von zahlreichen Brokern unterstützt, darunter ActiveMQ, RabbitMQ und Artemis. Typische Anwendungsfälle sind WebSocket-basierte Anwendungen oder Integrationen zwischen Webclients und Backendsystemen, bei denen Klartextkommunikation vorteilhaft ist.

Trotz seiner Lesbarkeit bringt STOMP Einschränkungen mit sich: Es bietet keine native Unterstützung für Quality-of-Service-Level wie in MQTT oder Transaktionskonzepte wie in AMQP. Auch fortgeschrittene Routing- oder Persistenzmechanismen sind brokerseitig zu realisieren. Für Szenarien mit textueller Nachrichtenstruktur und einfachen Anforderungen an Zuverlässigkeit und Performance ist STOMP jedoch eine geeignete Option.

### 2.1.2 Einordnung im Kontext dieser Arbeit

Da im Rahmen dieser Arbeit XML als Nachrichtenformat verwendet wird und eine gute Lesbarkeit sowie einfache Testbarkeit der Nachrichtenströme erforderlich sind, wäre der Einsatz von STOMP grundsätzlich denkbar. Aufgrund begrenzter Unterstützung moderner Protokollfeatures und fehlender nativer Integration mit MQTT5 oder JMS wurde STOMP im Evaluationsprozess jedoch nicht weiterverfolgt.

## 2.2 Features einer Message-Oriented Middleware

MOM bietet eine Vielzahl von Funktionen, die sie zu einer essenziellen Komponente in modernen, verteilten Systemarchitekturen machen. Ziel dieser Middleware ist es, die Kommunikation zwischen lose gekoppelten Systemkomponenten zuverlässig, flexibel und effizient zu gestalten [10].

Ein zentrales Merkmal ist die **asynchrone Kommunikation**, bei der Sender und Empfänger nicht gleichzeitig aktiv sein müssen. Nachrichten werden zwischengespeichert und erst verarbeitet, wenn der Empfänger verfügbar ist. Dies erhöht die Systemstabilität und Toleranz gegenüber Lastspitzen oder temporären Ausfällen [8]. Unterstützt wird dies typischerweise durch MQ oder Publish/Subscribe-Mechanismen, die eine entkoppelte Verteilung von Informationen ermöglichen.

Mit dem einher geht die **Zuverlässigkeit der Nachrichtenübertragung**. MOM-Systeme stellen sicher, dass Nachrichten nicht verloren gehen, doppelt verarbeitet werden oder in falscher Reihenfolge ankommen. Dies geschieht u. a. durch **Transaktionsunterstützung**, **Persistenzmechanismen** und **genaue Zustellgarantien** wie 'at least once', 'at most once' oder 'exactly once' [14].

Ein zentrales Leistungsmerkmal moderner MOM-Systeme ist ihre **Skalierbarkeit**. Diese bezeichnet die Fähigkeit eines Systems, bei steigender Last — sei es durch mehr Nachrichten, mehr Clients oder größere Nachrichtenvolumina — weiterhin performant und zuverlässig zu arbeiten. Skalierbarkeit kann dabei in mehreren Dimensionen betrachtet werden:

- Vertikale Skalierbarkeit (Scale-Up): Die Leistungssteigerung durch Zuweisung zusätzlicher Ressourcen auf einer einzelnen Instanz, z. B. mehr CPU, RAM oder schnellere Speichermedien. Diese Form ist oft durch physikalische Grenzen limitiert.
- Horizontale Skalierbarkeit (Scale-Out): Die Erweiterung des Systems durch das Hinzufügen weiterer Server oder Instanzen. Dies ist typisch für verteilte MOM-Systeme wie Apache Kafka oder RabbitMQ in Cluster-Konfigurationen [21].

- Elastizität: Die Fähigkeit, Ressourcen dynamisch zur Laufzeit basierend auf aktuellen Lastbedingungen automatisch zu skalieren — z. B. in Container- oder Cloud-Umgebungen (z. B. Kubernetes mit Helm-Deployments für Kafka-Broker) [10].

Skalierbarkeit lässt sich in der Praxis durch verschiedene **Kennzahlen und Metriken** quantifizieren, darunter:

- Durchsatz (Throughput): Anzahl verarbeiteter Nachrichten pro Sekunde (msg/s).
- Latenz: Zeit vom Absenden bis zum Eintreffen einer Nachricht beim Empfänger, gemessen in Millisekunden (ms).
- Verarbeitungszeit (Processing Time): Zeit, die eine Nachricht benötigt, um vollständig durch das System verarbeitet zu werden.
- Ressourcenauslastung: CPU-, RAM- und I/O-Verbrauch bei verschiedenen Lastniveaus.
- Fehlertoleranz unter Last: Anzahl verlorener Nachrichten bei bestimmten Ausfall- oder Überlastszenarien.

Systeme wie Apache Kafka erreichen z. B. mehrere hunderttausend bis Millionen Nachrichten pro Sekunde in produktiven Umgebungen, insbesondere durch Partitionierung und Replikation von Topics [21]. Auch Artemis lässt sich in Clustern mit Load-Balancing, Failover-Mechanismen und skalierbarer Message-Verteilung effizient betreiben, wobei die Performance stark von der Konfiguration der Paging-, Journal- und Speicheroptionen abhängt [3].

Im Hinblick auf **Sicherheit** bieten viele MOMs Funktionen wie TLS-verschlüsselte Kommunikation, rollenbasierte Zugriffskontrollen (RBAC) und Authentifizierungsmechanismen auf Basis moderner Standards wie OAuth 2.0 [10]. Diese Maßnahmen sind vor allem in sensiblen Industrie- oder Cloud-Umgebungen essenziell.

Ein weiteres Merkmal ist die **Interoperabilität**. MOM-Systeme unterstützen häufig offene Protokolle und Standards wie in 2.1 erwähnt, was eine einfache Integration heterogener Systeme erlaubt. Einige Systeme bieten zudem REST- oder gRPC-Schnittstellen, was die Anbindung an moderne Microservice-Architekturen erleichtert.

Moderne MOMs bieten zusätzlich **Monitoring- und Managementfunktionen**, die Administratoren Einblick in Systemzustände, Nachrichtenflüsse und Fehlersituationen ermöglichen. Diese Funktionen sind entscheidend für die Fehlerdiagnose und Systemoptimierung im Betrieb.

Zusammenfassend lassen sich folgende Kernfeatures einer MOM identifizieren:

- Asynchrone Kommunikation
- Nachrichtenpersistenz und Transaktionssicherheit
- Skalierbarkeit und Lastverteilung
- Sicherheitsmechanismen (TLS, Authentifizierung, Zugriffskontrolle)
- Interoperabilität durch Standardprotokolle
- Verwaltung, Überwachung und Fehlerbehandlung

MOMs weisen aber auch einige Herausforderungen auf. Ihr großer Umfang an Funktionen erhöht die Komplexität sowie die Anfälligkeit für Fehler bei unzureichender Konfigurierung. Auch das Monitoring oder Warten der Systeme kann bei großen Clustern zu Komplikationen führen.

Trotz dieser Herausforderung machen die vorherigen Eigenschaften MOM zu einem unverzichtbaren Bestandteil vieler IT-Systeme, insbesondere in verteilten oder hochautomatisierten Umgebungen.

## 2.3 State of the Art

Der aktuelle Stand der Technik im Bereich MOM ist geprägt durch eine Vielzahl hochentwickelter Plattformen und Technologien, die auf unterschiedliche Anwendungsfälle und Leistungsanforderungen ausgerichtet sind. Während traditionelle MOM-Systeme wie IBM MQ oder TIBCO Enterprise Message Service weiterhin in vielen Unternehmen im Einsatz sind, gewinnen moderne, cloud-native und Open-Source-basierte Lösungen zunehmend an Bedeutung [10].

Zu den marktführenden MOM-Technologien gehören unter anderem **Apache Kafka**, ein verteiltes Streaming-System, das ursprünglich von LinkedIn entwickelt wurde. Kafka ist auf hohe Durchsatzraten, horizontale Skalierbarkeit und Persistenz großer Datenmengen ausgelegt und eignet sich besonders für Event-Streaming und Datenanalyse in Echtzeit [21]. **RabbitMQ** ist ein weit verbreiteter Open-Source-Message-Broker, der das AMQP-Protokoll verwendet und besonders für Anwendungen mit komplexem Routing und hoher Integrationsdichte geeignet ist [30]. Ebenfalls zu erwähnen sind **ActiveMQ** und dessen performantere Weiterentwicklung **Artemis**, die insbesondere in Java-basierten Systemen aufgrund der JMS-Unterstützung weit verbreitet sind [3].

Weitere leichtgewichtige Vertreter wie **NATS** oder **ZeroMQ** fokussieren sich auf minimale Latenz und hohe Verfügbarkeit, was sie für IoT- und Microservice-Architekturen besonders attraktiv macht. Diese Systeme verzichten häufig auf komplexe Persistenzmechanismen und setzen auf Geschwindigkeit und Effizienz.

Ein signifikanter Trend in der Weiterentwicklung von MOM ist die tiefe Integration in Container- und Kubernetes-Umgebungen sowie die Bereitstellung als vollständig verwaltete Cloud-Dienste. Angebote wie **Amazon MQ**, **Azure Service Bus** oder **Google Cloud Pub/Sub** bieten Out-of-the-Box-Lösungen für skalierbares Messaging mit integrierter Sicherheit, Monitoring und Hochverfügbarkeit [10].

Moderne MOM-Systeme unterstützen häufig hybride Kommunikationsmodelle, die neben klassischem asynchronem Messaging auch Event-Streaming und synchrone Kommunikationsformen integrieren. Zusätzlich werden umfangreiche Sicherheitsmechanismen wie TLS-Verschlüsselung, OAuth 2.0-Authentifizierung und rollenbasierte Zugriffskontrollen implementiert, um den Anforderungen an Datenschutz, Compliance und Sicherheit gerecht zu werden [14].

Insgesamt lässt sich festhalten, dass sich MOM-Technologien von klassischen, monolithischen Lösungen hin zu modularen, hochperformanten und cloudfähigen Systemen entwickelt haben, die zentrale Bausteine moderner verteilter IT-Architekturen darstellen.

## 2.4 Abgrenzung zu Event-Streaming am Beispiel Apache Kafka

Ein besonders weit verbreitetes System im Bereich der asynchronen Kommunikation ist *Apache Kafka*. Kafka wurde ursprünglich bei LinkedIn entwickelt und 2011 als Open-Source-Projekt in die Apache Software Foundation eingebracht [21]. Im Gegensatz zu klassischen Message-Oriented-Middleware-Systemen handelt es sich bei Kafka jedoch primär um eine verteilte Event-Streaming-Plattform, deren Architektur stark auf die persistente Speicherung und Verarbeitung von Datenströmen ausgelegt ist. Nachrichten (Events) werden dabei in logbasierten Strukturen dauerhaft abgelegt, wodurch ein besonders hoher Durchsatz sowie eine effiziente Verarbeitung großer Datenmengen möglich ist [4].

Kafka eignet sich insbesondere für Anwendungen, bei denen kontinuierlich große Datenvolumina generiert und in Echtzeit analysiert werden müssen, wie etwa bei Monitoring- oder Analytics-Szenarien in Cloud- und Big-Data-Umgebungen. In diesen Einsatzbereichen bietet Kafka eine nahezu unbegrenzte horizontale Skalierbarkeit sowie eine hohe Fehlertoleranz durch Replikationsmechanismen.

Für die vorliegende Arbeit und den spezifischen Anwendungsfall der hochautomatisierten Halbleiterfertigung ist Kafka jedoch nicht die geeignete Wahl. Der hohe Administrationsaufwand, die vergleichsweise komplexe Systemarchitektur und die im Vergleich zu leichtgewichtigeren MOMs höhere Latenz wirken sich nachteilig aus. Zudem stehen für die Integration in bestehende Systeme insbesondere die Unterstützung standardisierter Schnittstellen wie JMS oder MQTT im Vordergrund – ein Funktionsbereich, der von Kafka nur eingeschränkt abgedeckt wird. Somit wird Kafka im Rahmen dieser Untersuchung nicht als Kandidat für die zukünftige Middleware-Lösung berücksichtigt.

## 2.5 Status Quo in X-FAB

Wie bereits in Abschnitt 1.2 erläutert, kommt in der X-FAB Dresden derzeit der TIBCO Message Bus als zentrale Kommunikationsplattform zum Einsatz. Die Infrastruktur besteht aus zwei separaten Serverinstanzen, die im Clusterverbund betrieben werden. Ein Loadbalancer wird dabei bewusst nicht verwendet, da es in Failover-Szenarien wiederholt zu Verbindungsproblemen seitens der Clients kam.

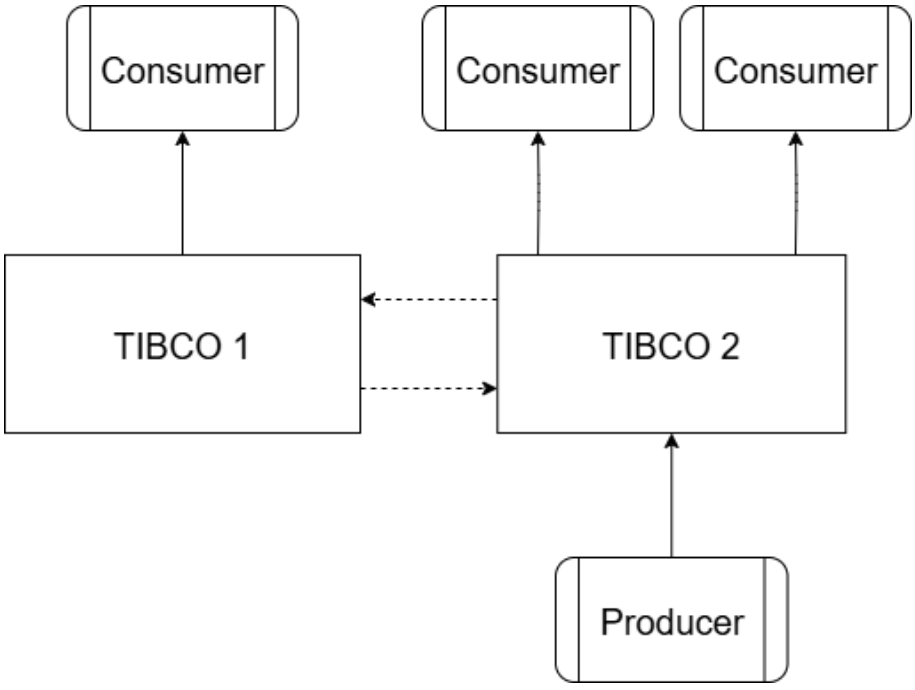



Abbildung 2.1: TIBCO Struktur

Der TIBCO-Bus nutzt Multicast ausschließlich für die initiale Erkennung und Kommunikation zwischen den am Cluster beteiligten Brokern. Über das Multicast-Verfahren können sich die verfügbaren TIBCO-Broker im Netzwerk selbstständig entdecken und ihren aktuellen Status austauschen. Dies ermöglicht eine dynamische Clusterbildung und die automatische Auswahl eines verfügbaren Brokers bei Ausfall eines Knotens.

Die eigentliche Übertragung der Nachrichten erfolgt hingegen klassisch über TCP. Publisher senden ihre Nachrichten direkt an einen aktiven Broker, der diese wiederum über dedizierte TCP-Verbindungen an alle Subscriber weiterleitet, die das entsprechende Subject abonniert haben. Die gesamte Verteilung der Nutzdaten geschieht also nicht via Multicast, sondern über zuverlässige Punkt-zu-Punkt-Verbindungen (unicast) zwischen dem Broker und den angeschlossenen Clients.

Eine Besonderheit der TIBCO-Implementierung liegt in deren Resilienzkonzept: Der Server ist so konfiguriert, dass er sich bei internen Problemen automatisch neu startet. Eine vollständige Abschaltung des Systems ist ausschließlich durch physikalisches Trennen der Stromversorgung möglich. Diese Eigenschaft erhöht zwar die Verfügbarkeit, erschwert jedoch gezielte Wartungsmaßnahmen oder geplante Systemstillstände.

## 2.6 Übersicht erwähnter Messaging-Standards

Tabelle 1 fasst die im Rahmen dieser Arbeit relevanten Messaging-Standards sowie ausgewählte Implementierungen kompakt zusammen und verdeutlicht die Abgrenzung zwischen Standard und konkreter Umsetzung. Einige Implementierungen nutzen auch verschieden Standards, wobei hier immer nur eine Auswahl gezeigt wird.

Tabelle 1: Übersicht der in dieser Arbeit behandelten Messaging-Standards und deren Implementierungen

| **Standard** | **Beschreibung** | **Kommunikations-modelle** | **Beispiel-Implementierungen** |
|---|---|---|---|
| **JMS** | Standardisierte Java-API für asynchrone Nachrichtenübertragung; abstrahiert von der konkreten Middleware | Point-to-Point, Publish/Subscribe | Apache ActiveMQ, Apache ActiveMQ Artemis, IBM MQ, TIBCO EMS |
| **MQTT** | Leichtgewichtiges Protokoll für M2M- und IoT-Kommunikation mit geringer Bandbreiten- und Ressourcenanforderung | Publish/Subscribe | Eclipse Mosquitto, HiveMQ, EMQX |
| **AMQP** | Offener, plattformübergreifender Standard mit garantierter Zustellung und flexibler Nachrichtensteuerung | Point-to-Point, Publish/Subscribe, Request/Response | RabbitMQ, Apache Qpid, Azure Service Bus |
| **STOMP** | Einfaches textbasiertes Protokoll mit HTTP-ähnlicher Nachrichtenstruktur; besonders geeignet für lesbare Formate (z. B. XML, JSON) | Point-to-Point, Publish/Subscribe (brokerabhängig) | Apache ActiveMQ, RabbitMQ, Apache ActiveMQ Artemis |
| **Eigener Standard** | Event-Streaming-Plattform für hochskalierbare, persistente Nachrichtenströme in Echtzeit | Log-basierte Event-Streams | Apache Kafka, Confluent Kafka |
| — | Lightweight-Messaging-System mit Fokus auf minimale Latenz und hohe Verfügbarkeit | Publish/Subscribe | NATS, ZeroMQ |

# 3 Anforderungsanalyse

## 3.1 Lastanalyse

Um ein besseres Verständnis über das aktuelle Lastaufkommen auf dem bestehenden Message Bus zu erhalten, wurde mithilfe eines von der SYSTEMA GmbH entwickelten Monitoring-Tools eine einstündige Aufzeichnung durchgeführt. Das Tool erfasst dabei eine Vielzahl von Parametern zu jeder übertragenen Nachricht, darunter die Gesamtanzahl je Nachrichtentyp, die durchschnittliche Nachrichtenfrequenz (Nachrichten pro Sekunde) sowie die mittlere Nachrichtengröße. Die erfassten Daten lassen sich im Anschluss als Excel-Datei exportieren und detailliert auswerten.

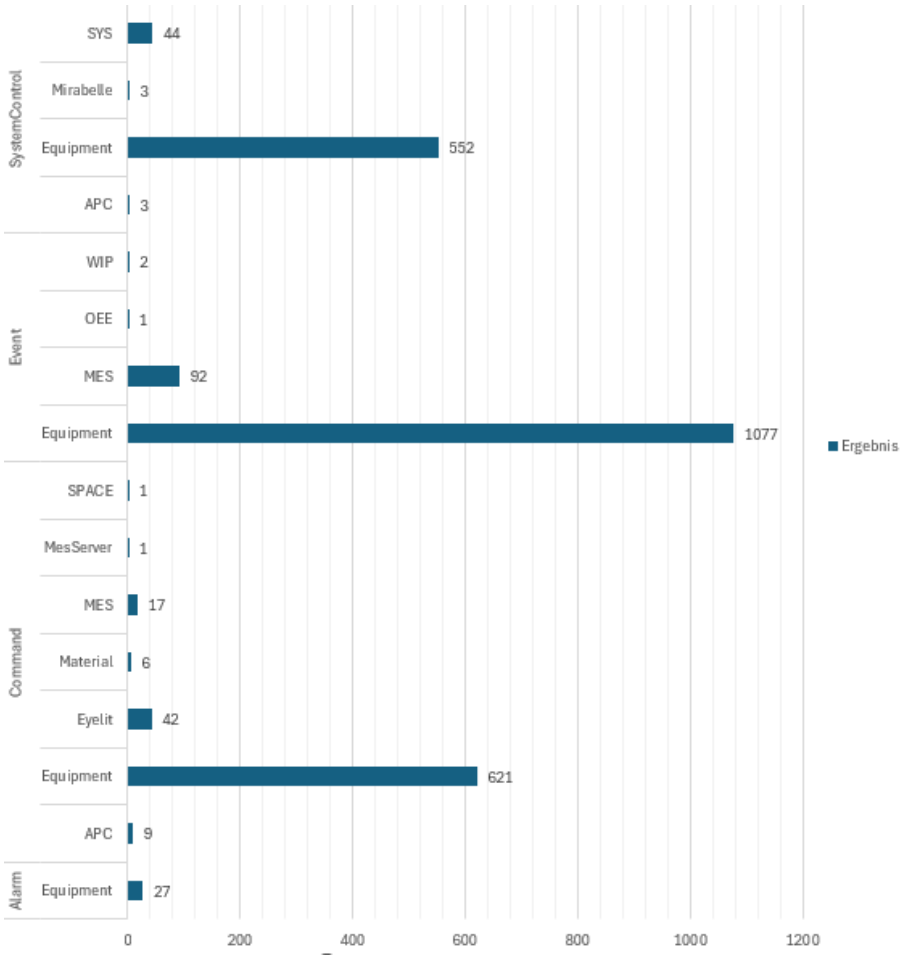



Abbildung 3.1: Topic Verteilung

Wie in Abbildung 3.1 ersichtlich ist, entstammt der Großteil der übertragenen Nachrichten dem Topic-Bereich *Equipment*. Dieser umfasst sämtliche Komponenten der Produktionsumgebung, insbesondere die verschiedenen Fertigungsanlagen, die eine Vielzahl unterschiedlicher Nachrichten – beispielsweise zu Statusänderungen – erzeugen und versenden.

Insgesamt sind somit rund 2.500 verschiedene Nachrichtentypen im produktiven Einsatz, die sich hinsichtlich Frequenz, Umfang und Inhalt deutlich unterscheiden.

Abbildung 3.2 gibt eine Übersicht über das Nachrichtenaufkommen nach Größe und Anzahl. Es zeigt sich deutlich, dass der überwiegende Teil der Nachrichten eine geringe Datenmenge aufweist, jedoch mit hoher Frequenz übertragen wird. Gleichzeitig ist zu erkennen, dass vereinzelt auch Nachrichten mit deutlich größerem Umfang auftreten, die im Vergleich seltener gesendet werden, jedoch eine relevante Belastung für die Middleware-Infrastruktur darstellen können.

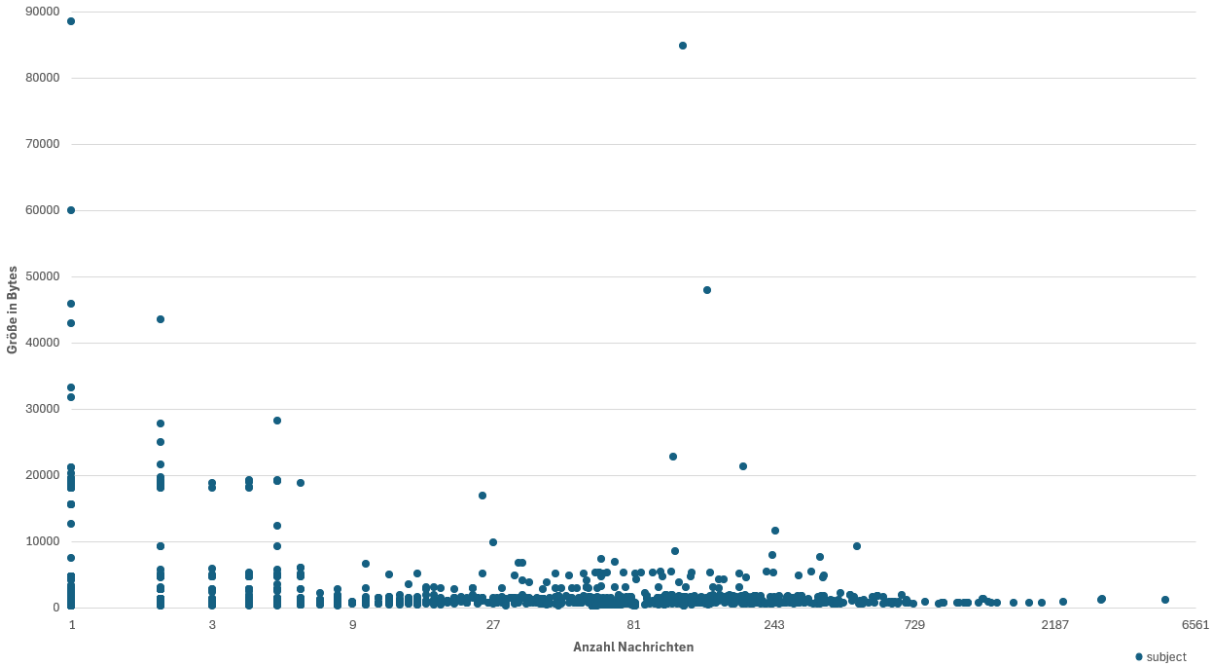



Abbildung 3.2: Nachrichten

Die aggregierte Auswertung der einstündigen Aufzeichnung über alle rund 2.500 Topics hinweg liefert ein umfassendes Bild des Gesamtaufkommens auf dem Message Bus. Während des Erfassungszeitraums wurden insgesamt **232.973 Nachrichten** verarbeitet, was einer durchschnittlichen Übertragungsrate von etwa **65 Nachrichten pro Sekunde** entspricht. Die durchschnittliche Größe einer einzelnen Nachricht betrug **1.940 Byte**, wobei eine durchschnittliche Übertragungsgröße von **85.000 Bytes pro Sekunde** festgestellt wurde. Über die gesamte Messdauer summiert sich das Gesamtübertragungsvolumen auf etwa **299 Megabyte**.

Diese Kennzahlen verdeutlichen, dass auch bei einer Vielzahl kleiner und hochfrequent gesendeter Nachrichten ein nicht zu vernachlässigendes Datenvolumen generiert wird. Vor dem Hintergrund von Echtzeitanforderungen und einer potenziell weiter steigenden Anzahl von Nachrichtentypen und -frequenzen ist eine leistungsfähige, skalierbare Middleware-

Infrastruktur entscheidend, um eine stabile und effiziente Kommunikation innerhalb der Fertigungsumgebung sicherzustellen.

Das Monitoring-Tool stellt neben den aggregierten Werten auch weitere relevante Messdaten zur Verfügung. Besonders aufschlussreich ist dabei die Betrachtung von Lastspitzen (Peaks): Der maximale Durchsatz an Nachrichten wurde mit **390 Nachrichten pro Sekunde** gemessen, während der höchste gemessene Datenstrom bei **1.350.000 Byte pro Sekunde** lag. Die größte Nachricht benötigte **88.000 Bytes** Speicherplatz. Die Messwerte wurden am 16. Mai 2024 gegen 16 Uhr unter Bedingungen einer nahezu vollständig ausgelasteten Fertigungslinie erhoben und eignen sich daher in besonderem Maße für einen aussagekräftigen Leistungsvergleich.

## 3.2 Mindestanforderung

Ein zukünftiges Bus-System muss mindestens in der Lage sein, diese Spitzenwerte stabil zu verarbeiten. Zusätzlich sollten weitere funktionale Anforderungen berücksichtigt werden, um eine hohe Verfügbarkeit und Zukunftsfähigkeit der Systemarchitektur zu gewährleisten:

- Durchführung von Updates einzelner Knoten ohne Betriebsunterbrechung (Zero Downtime Deployment)
- Automatischer Failover-Mechanismus: Wiederverbindung des Clients zu einem alternativen Server bei Ausfall des ursprünglichen Knotens
- Unterstützung eines MQTT 5-kompatiblen Brokers

## 3.3 Bedeutung von MQTT 5 im Unternehmenskontext

MQTT 5 stellt einen bedeutenden Fortschritt gegenüber der Vorgängerversion 3.1.1 dar und bietet zahlreiche Erweiterungen, die insbesondere für moderne IoT-Anwendungen unerlässlich sind. In einem Unternehmenskontext, in dem vermehrt verteilte Sensorik, intelligente Geräte sowie performante Kommunikationsstrukturen benötigt werden, ermöglicht MQTT 5 eine robustere und gleichzeitig flexibel anpassbare Nachrichtenvermittlung.

Ein zentraler Vorteil liegt in der verbesserten Unterstützung für IoT-Anwendungen. MQTT 5 erlaubt unter anderem die Verwendung von sogenannten *User Properties*, mit denen benutzerdefinierte Metadaten an Nachrichten angehängt werden können. Diese können beispielsweise genutzt werden, um Routing-Entscheidungen zu vereinfachen oder zusätzliche

Kontextinformationen zu übertragen [28]. Damit lassen sich komplexere Applikationslogiken abbilden, ohne den Payload verändern zu müssen. Funktionen wie das *Reason Code System* erlauben eine feinere Fehlerbehandlung und ermöglichen eine präzisere Rückmeldung bei fehlgeschlagenen Operationen [16]. Damit wird die Entwicklung robuster Applikationen deutlich vereinfacht.

Ein zentraler Vorteil von MQTT 5 besteht darin, dass es auch in weniger technisch geprägten Abteilungen problemlos eingesetzt werden kann. Dank klar strukturierter Protokollfunktionen, wie etwa benutzerdefinierter Eigenschaften zur Kennzeichnung von Nachrichten oder standardisierter Fehlermeldungen, ermöglicht es den Mitarbeitenden die Nutzung eines gemeinsamen Kommunikationssystems, ohne dass tiefgreifendes Wissen über die interne Architektur des Message Bus erforderlich ist. Dadurch können verschiedene Unternehmensbereiche effizient zusammenarbeiten und bestehende Systeme leichter integriert werden.

Weitere zentrale Features von MQTT 5, die in der Vorgängerversion fehlen, sind unter anderem:

- **Session Expiry Interval:** Ermöglicht zeitlich begrenzte Sessions für temporäre Geräte, ohne manuelles Session-Management.
- **Message Expiry Interval:** Nachrichten können mit einem Ablaufdatum versehen werden, was insbesondere bei flüchtigen Sensorwerten nützlich ist.
- **Shared Subscriptions:** Mehrere Clients können eine Subscription teilen, was die Lastverteilung bei großen Systemen erleichtert.
- **Request/Response Pattern:** Neue Möglichkeiten für synchrone Kommunikation innerhalb eines ansonsten asynchronen Protokolls.

## 3.4 Wünschenswerte Anforderung

Ergänzend zu diesen Kernanforderungen existieren optionale, aber wünschenswerte Funktionalitäten, insbesondere im Hinblick auf Analysefähigkeit, Betriebssicherheit und Benutzerfreundlichkeit:

- Ereignisaufzeichnung mit zeitlicher Sortierung zur späteren Auswertung (Event Logging)
- Persistente Nachrichtenwarteschlangen für nicht erreichbare Clients (Message Store-and-Forward)

- Unterstützung zur Verschlüsselung mithilfe von Zertifikaten (bzw. verschlüsselter Kommunikation )
- Mechanismen zur dynamischen Lastverteilung (Load Balancing)

## 3.5 Anforderungskatalog

Anhand des Anforderungskatalog kann eine Eindeutige Bewertung am Ende vorgenommen werden. Dabei können in jeder Kategorie bis zu 5 Punkte erreicht werden, welche mit der Gewichtung multipliziert werden.

Tabelle 2: Anforderungskatalog für die Evaluierung der MOM-Systeme

| **Kategorie** | **Anforderung** | **Gewichtung (1–5)** |
|---|---|---|
| Funktional | Unterstützung von JMS | 5 |
| Funktional | Unterstützung von MQTT 5 | 5 |
| Funktional | Unterstützung von Publish/Subscribe und Queue-Modell | 4 |
| Performance | Nachrichtenlatenz <10 ms bei Produktionslast | 5 |
| Performance | Verarbeitung von >400 Nachrichten/s (Peak 390 gemessen) | 5 |
| Resilience / Availability | Failover ohne Datenverlust | 5 |
| Resilience / Availability | Automatische Wiederverbindung von Clients | 4 |
| Resilience / Availability | Clustering und horizontale Skalierung | 4 |
| Usability | Einfache Konfiguration (z. B. über Konfigurationsdateien oder GUI) | 3 |
| Usability | Integration in bestehende Tools und Monitoring | 3 |
| Usability | Gute Dokumentation und Community-Support | 2 |

## 3.6 Nachrichtenformat

In der X-FAB GmbH wurde Extensible Markup Language (XML) als zentrales Nachrichtenformat für den Austausch zwischen verteilten Komponenten festgelegt. Auch in der vorliegenden Arbeit wurde sich für XML entschieden. Diese Entscheidung erfolgte bewusst unter Abwägung alternativer Formate wie YAML oder JSON. Im Folgenden werden die Eigenschaften, Vorteile und Herausforderungen von XML ausführlich diskutiert sowie alternative Formate gegenübergestellt und ihre Eignung für unseren Anwendungsfall bewertet.

### 3.6.1 Grundlegende Eigenschaften von XML

XML ist ein textbasiertes, selbstbeschreibendes Datenformat, das nach der Spezifikation der W3C Working Group definiert ist [34]. Wesentliche Merkmale sind:

- **Hierarchische Struktur:** XML-Dokumente bestehen aus verschachtelten Elementen, die eine Baumstruktur („Document Object Model“, DOM) bilden. Diese Struktur erlaubt die Abbildung komplexer, verschachtelter Datenschemata, wie sie in industriellen Steuerungsdaten häufig vorkommen.
- **Selbstbeschreibend:** Jedes Element ist durch Tags eindeutig benannt, wodurch semantische Bedeutung direkt im Dokument verankert ist. Dies erleichtert das Verständnis und die Nachvollziehbarkeit der Daten auch für ungeübte Anwender.
- **Schema-basierte Validierung:** Mit Dokumenttypdefinitionen (DTD) oder XML Schema Definition Language (XSD) können die Struktur, Datentypen und cardinality (Vorkommenshäufigkeiten) von Nachrichten formal definiert und automatisch validiert werden [33].
- **Namespaces:** Durch XML Namespaces lassen sich Elemente und Attribute aus verschiedenen Anwendungsdomänen eindeutig voneinander trennen, was insbesondere in heterogenen Systemlandschaften Konflikte bei gleicher Namensverwendung vermeidet.
- **Transformation und Abfrage:** Technologien wie XSLT (Extensible Stylesheet Language Transformations) und XPath erlauben leistungsfähige Transformationen und gezielte Abfragen auf XML-Dokumenten, wodurch Datenflüsse flexibel angepasst und analysiert werden können.
- **Ausgereiftes Werkzeug-Ökosystem:** Für nahezu jede Programmiersprache existieren Bibliotheken zum Parsen (DOM, SAX, StAX), Validieren und Transformieren von XML, wodurch eine einfache Integration in bestehende Softwarelandschaften gewährleistet ist.

### 3.6.2 Vorteile für die hochautomatisierte Fertigung

In der Halbleiterproduktion sind Datenformate gefordert, die sowohl Strukturkomplexität als auch strenge Konsistenz garantieren. XML bietet hier:

- **Robuste Validierung:** Durch XSD-Validierung können strukturbedingte Fehler bereits auf Transportebene abgefangen werden – ein entscheidender Faktor, um fehlerhafte Steuerbefehle oder Messwerte frühzeitig zu erkennen.
- **Interoperabilität:** XML ist breit akzeptiert und standardisiert, sodass unterschiedliche Systeme (z. B. MES, SCADA, Datenanalyseplattformen) nahtlos miteinander kommunizieren können.
- **Langfristige Wartbarkeit:** Die Selbstbeschreibungsfähigkeit und klare Schema-Definition erleichtern die Dokumentation und spätere Anpassung der Nachrichtenstrukturen.

### 3.6.3 Herausforderungen und Optimierungsansätze

Trotz seiner Stärken bringt XML auch typische Nachteile mit sich:

- **Verbosity (Übertragungs-Overhead):** XML-Dokumente sind durch Tags oft deutlich größer als binäre Formate, was zu höherem Bandbreiten- und Speicherbedarf führt.
- **Parsing-Kosten:** Die textuelle Natur erfordert aufwendiges Parsen, was in ressourcenbeschränkten Geräten zu Performance-Engpässen führen kann.
- **Komplexität von Namespaces und Schemata:** Ungeübte Entwickler können bei Namespaces und XSD-Features leicht Fehler einschleusen.

Zur Milderung dieser Nachteile können Kompressionsverfahren (z. B. GZIP) oder Streaming-basierte Parser (StAX) eingesetzt werden, um Speicher- und CPU-Belastung zu reduzieren.

### 3.6.4 Alternative Nachrichtenformate

Im folgenden Vergleich werden vier gängige Alternativen kurz vorgestellt und hinsichtlich Eignung für unseren Anwendungsfall bewertet:

**JavaScript Object Notation (JSON)** Ein leichtgewichtiges, textbasiertes Format mit einfacher Syntax [9]. JSON verzichtet auf formale Typ-Validierung (außer optional via JSON Schema) und unterstützt keine Namespaces. Obwohl JSON wegen seiner Kompaktheit und guten Lesbarkeit in Web-APIs weit verbreitet ist, fehlen die robusten Schema- und Transformationsmechanismen von XML, weshalb es für unsere strukturkritischen Produktionsnachrichten weniger geeignet ist.

**Protocol Buffers (Protobuf)** Ein von Google entwickeltes, binäres Format mit starker Schema-Bindung [15]. Protobuf bietet extrem geringe Nachrichtenvolumina und hohe Parsing-Geschwindigkeiten, erfordert jedoch das Vorhalten und Kompilieren von '.proto'-Definitionen. Die starre Kopplung von Schema und Daten erschwert dynamische Erweiterungen und kurzfristige Schema-Änderungen in produktiven Systemen.

**Apache Avro** Ein weiteres binäres, schema-basiertes Format mit Unterstützung für dynamische Schemas [2]. Die Daten werden zusammen mit dem Schema transportiert, was Flexibilität bietet, jedoch einen zusätzlichen Overhead verursacht und ein zentrales Schema-Repository voraussetzt.

**YAML Ain't Markup Language (YAML)** Ein textbasiertes, menschenlesbares Format, das auf Einrückungen basiert [37]. YAML eignet sich für Konfigurationsdateien, aber bei sehr tief verschachtelten Strukturen oder großen Nachrichten kann die fehleranfällige Syntax Parsing-Probleme verursachen und die maschinelle Verarbeitung erschweren.

### 3.6.5 Fazit

Für die Anforderungen der hochautomatisierten Halbleiterfertigung erweist sich XML als das geeignetste Format, da es eine ausgewogene Balance zwischen Strukturkomplexität, Validierbarkeit und weitreichender Tool-Unterstützung bietet. Spezialisierte, binäre Formate wie Protobuf oder Avro sind zwar in Hinblick auf Effizienz überlegen, jedoch für dynamische Schema-Anpassungen und strikte Validierungsanforderungen weniger flexibel. JSON und YAML kommen als leichtgewichtige Alternativen aufgrund fehlender Built-In-Validierung und Transformationsstandards nicht in Frage.

# 4 Analyse unterschiedlicher MOMs

## 4.1 Vorauswahl

Im Folgenden werden verschiedene MOM-Lösungen betrachtet und für eine detaillierte Analyse eingegrenzt. Die SYSTEMA GmbH hat im Vorfeld eine Vorauswahl getroffen, welche Messaging-Plattformen gemäß den aktuellen und zukünftigen Anforderungen als potenzieller FAB-Message-Bus in Frage kommen. Eine Auflistung der geprüften Systeme ist in Abbildung 4.1 dargestellt. Die beiden Protokolllösungen MQTT3 und 5 wurden ebenfalls in die Auswahl einbezogen. Obwohl diese nicht direkt als Messaging-Systeme betrachtet werden können, erweist sich ihre Berücksichtigung als sinnvoll, da sie im Kontext von Nachrichtensystemen zahlreiche Vorteile bieten.

| | | | |
|---|---|---|---|
| ActiveMQ | Apollo | Artemis | BMQ |
| FTL | Kafka | MBX | MQSeries-Websphere |
| MQTT3 | MQTT5 | MantaRay | NATS |
| SAP NetWeaver | OpenJMS | RabbitMQ | Spread |
| TIB / Rendezvous | TIBCO EMS | WSO2 | WebLogic |

Abbildung 4.1: Vorauswahl möglicher Message-Bus-Lösungen [13]

Einige Systeme konnten frühzeitig über SYSTEMA ausgeschlossen werden, da sie entweder konzeptionell nicht dem Paradigma einer MOM entsprechen oder aus technologischer Sicht nicht zukunftsfähig sind. Dazu zählen:

- *SAP NetWeaver*, *TIBCO EMS* und *WSO2*: Diese Plattformen sind keine klassischen MOMs, sondern integrierte Middleware-Lösungen mit anderem Fokus.
- *Apache Kafka*: Obwohl weit verbreitet, handelt es sich hierbei um eine Event-Streaming-Plattform und nicht um eine traditionelle MOM im engeren Sinne. Detaillierte Analyse siehe Kapitel 2.4.
- *Apollo*, *BMQ*, *MBX*, *MQSeries*, *MantaRay*, *OpenJMS* und *WebLogic*: Diese Produkte gelten als veraltet und sind heute bestenfalls noch als Legacy-Komponenten in Betrieb, aber für neue Infrastrukturen nicht geeignet.

- Weitere Systeme wie *FTL* (Konfigurationsprobleme), *MQTT 3.1* (veraltet, nicht skalierbar), *RabbitMQ* (eingeschränkte Bedienbarkeit) und *Spread* (Instabilität) erfüllen die funktionalen Anforderungen nicht ausreichend und wurden daher ebenfalls ausgeschlossen.
- *ActiveMQ* und *TIB*: Beide Systeme gelten als State-of-the-Art im klassischen MOM-Spektrum, sind jedoch aufgrund strategischer oder technischer Einschränkungen keine geeigneten Kandidaten für eine langfristige Neuausrichtung.

Nach dieser ersten Auswahl verbleiben die folgenden drei Systeme für eine vertiefte Analyse:

- **Artemis**
- **MQTT Version 5**
- **NATS**

Diese Systeme werden im weiteren Verlauf hinsichtlich ihrer Leistungsfähigkeit, Zukunftssicherheit und Kompatibilität mit den definierten Anforderungen untersucht.

## 4.2 Vertiefte Analyse

Nachfolgend werden die vorher ausgewählten Systeme genauer betrachtet und sowohl miteinander als auch mit den bestehenden Anforderungen verglichen.

### 4.2.1 NATS

NATS ist eine vergleichsweise junge MOM-Lösung, deren erste stabile Version im Jahr 2011 von Synadia Communications Inc. veröffentlicht wurde [18]. Konzipiert als leichtgewichtige und hochverfügbare Messaging-Plattform, hat NATS seitdem eine Reihe signifikanter funktionaler Erweiterungen und Stabilitätsoptimierungen erfahren. Aktuellen Erhebungen zufolge setzen weltweit rund 1.670 Unternehmen NATS in ihren verteilten Systemlandschaften ein [19].

Die Architektur von NATS zeichnet sich durch hohe Fehlertoleranz und geringe Störanfälligkeit aus. So ist die Zahl dokumentierter Produktionsstörungen minimal, was auf robuste Implementierungen von Cluster- und Selbstheilungsmechanismen zurückzuführen ist [18]. Einschränkungen ergeben sich primär in Szenarien mit strikten Anforderungen an die Nachrichtenreihenfolge (Order-Sensitivity) sowie bei der Übertragung sehr großer Nachrichtenvolumina – Befunde, die auch für andere MOM-Systeme typisch sind [18].

Ein zentrales Gestaltungsmerkmal ist der geringe Ressourcenverbrauch: Eine Docker-Instanz des NATS-Servers belegt lediglich etwa 3 MB Speicher [22]. Trotz dieser Kompaktheit erreicht NATS Verarbeitungsraten von bis zu 11 Mio. Nachrichten pro Sekunde bei einer durchschnittlichen Nachrichtengröße von mehr als 1 KB [23]. Nachrichten bleiben dabei unverändert ("zero-copy") und werden nicht in zusätzliche Byte-Streams umgewandelt.

Die horizontale Skalierbarkeit von NATS basiert auf einer mehrstufigen Cluster-Architektur. Einzelne Server lassen sich zu Clustern und Clustern wiederum zu Super-Clustern zusammenfassen. Neue Knoten können im laufenden Betrieb hinzugefügt werden; das System justiert automatisch die Lastverteilung und gewährleistet so durchgängig hohe Verfügbarkeit [24]. Zusätzlich implementiert NATS Selbstheilungsroutinen, die bei Netzwerkausfällen oder Verbindungsabbrüchen eine automatische Reorganisation der Clusterstruktur initiieren.

NATS-Clients existieren für über 32 Programmiersprachen [26] und unterstützen sowohl Publish/Subscribe- als auch Request/Reply-Muster. Zur Laufzeit kann zwischen verschiedenen Zustellungsmodi gewählt werden:

- *Batch Delivery*: gebündelte Übertragung mehrerer Nachrichten,
- *Sequential Delivery*: sukzessive Zustellung einzelner Nachrichten,
- *Real-Time Delivery*: unmittelbar priorisierte Übermittlung mit minimaler Latenz.

Insbesondere der Real-Time-Modus ist für Anwendungen relevant, in denen exakte Timing-Vorgaben und geringe Verzögerungen eine zentrale Rolle spielen (z. B. in Regelungs- oder Monitoring-Szenarien).

Abschließend bietet NATS ein umfassendes Sicherheitsmodell. Neben klassischer Authentifizierung via Benutzername/Passwort lassen sich feingranulare Berechtigungsregeln für einzelne Topics (Subjects) definieren. Änderungen an der Zugriffskontrolle können ohne Systemneustart ausgerollt werden, wodurch der Zero-Downtime-Betrieb gewährleistet bleibt [25].

Trotz der genannten Vorteile erfüllt NATS derzeit nicht alle funktionalen Anforderungen, da bislang ausschließlich das Protokoll MQTT Version 3.1.1 unterstützt wird. Die Unterstützung für MQTT 5 wurde zwar bereits im Jahr 2022 im Rahmen eines Feature-Requests auf der offiziellen GitHub-Plattform vorgeschlagen, jedoch ist seither keine konkrete Weiterentwicklung erfolgt. Seitens der Entwickler besteht aktuell keine Nachfrage für dieses Feature und andere Themen haben höhere Priorität. [12] Aus diesem Grund ist derzeit nicht absehbar, ob und wann eine Integration von MQTT 5 in NATS vorgesehen ist.

### 4.2.2 MQTT 5

MQTT 5, formell spezifiziert 2019 durch das OASIS MQTT Technical Committee [28], adressiert zentrale Limitationen von MQTT 3.1.1 und erweitert das Protokoll um Mechanismen, die für den Einsatz in großskaligen IoT- und Industrie-4.0-Szenarien unerlässlich sind.

Bereits in der Standardisierung wurde besonderer Wert auf *Fehlerdiagnose* und *Feinsteuerung* gelegt. So führen die neuen „Reason Codes“ in ACK- und DISCONNECT-Paketen eine differenzierte Rückmeldung über Fehlerszenarien ein, was eine gezielte Automatisierung von Recovery-Strategien erlaubt [28]. Ergänzt wird dies durch das „Enhanced Authentication Framework“, das Challenge-Response-Abläufe standardisiert und damit die sichere Geräte-Authentifizierung selbst in untrusted Umgebungen verbessert [20].

Für das Sitzungsmanagement ermöglicht MQTT 5 nun das Setzen eines *Session Expiry Interval*, über das Clients definieren, wie lange ihr Zustand nach einer Trennung auf dem Broker vorgehalten wird. In Verbindung mit *Topic Aliases* zur Reduktion des Header-Overheads lassen sich so Ressourcen auf Broker-Seite effizienter nutzen und die Netzlast senken [28].

Ein weiterer Schwerpunkt liegt auf der *Skalierbarkeit*. Shared Subscriptions standardisieren das Load-Balancing über mehrere Client-Instanzen hinweg, und das Flow-Control-Feature erlaubt es subscribern seine momentane Empfangskapazität dem zugehörigen publisher über den Broker mitzuteilen. Beides zusammen verhindert Überlastsituationen und verbessert die Durchsatzfähigkeit in Clustern signifikant [32]. Performance-Messungen zeigen, dass MQTT 5-Broker in geeigneter Infrastruktur leicht Hunderttausende von Nachrichten pro Sekunde skalieren können.

Zur Laufzeitüberwachung und Analyse lassen sich *User Properties* in jedes Paket einbetten, wodurch Metadaten wie Zeitstempel, Priorität oder IoT-Sensordaten direkt transportiert werden. Diese Feingranularität unterstützt fortgeschrittene Monitoring- und Auditing-Szenarien, ohne separate Kanäle zu benötigen [28].

Schließlich verbessert MQTT 5 das Request/Response-Pattern durch definierte Response-Topics und *Correlation Data*, was den Entwurf bidirektionaler Microservice-Architekturen erleichtert. Sicherheitsrelevante Erweiterungen wie *Message Expiry* (TTL) und das Enhanced Authentication Framework wurden in Forschungsarbeiten auf ihre Robustheit untersucht und validiert [1].

Insgesamt macht MQTT 5 durch diese Erweiterungen einen deutlichen Schritt von einem rein leichtgewichtigen IoT-Protokoll hin zu einer Basis für eine Allzweck-Middleware,

die sowohl in hochdynamischen Automatisierungsumgebungen als auch in kritischen Industrieanwendungen eingesetzt werden kann.

### 4.2.3 Artemis

Apache ActiveMQ Artemis ist eine moderne, leistungsfähige und asynchron arbeitende Message-Oriented Middleware, die als Nachfolger von ActiveMQ Classic entwickelt wurde. Die modulare Architektur unterstützt eine Vielzahl etablierter Messaging-Protokolle, darunter AMQP, MQTT, STOMP und OpenWire. Artemis ermöglicht sowohl synchrone als auch asynchrone Kommunikation und beherrscht gängige Messaging-Muster wie Publish/Subscribe und Point-to-Point.

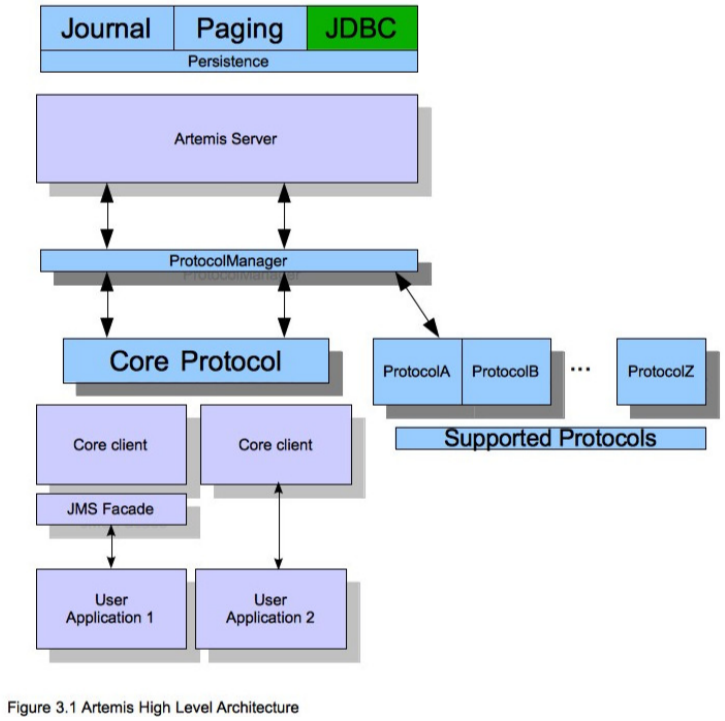



Abbildung 4.2: Architektur von Apache ActiveMQ Artemis [5]

**Architekturüberblick** Kern des Systems ist der Artemis-Broker, der gemeinsam mit einem Protokollmanager und einem Core-Message-Bus für eine effiziente und zuverlässige Nachrichtenverarbeitung sorgt. Zur Sicherstellung der Persistenz können Nachrichten wahlweise über ein performantes Journaling-System im Dateisystem oder über JDBC-basierte Datenbanken gespeichert werden. Bei hoher Last oder temporären Speicherengpässen nutzt Artemis einen Paging-Mechanismus, um den Nachrichtendurchsatz stabil zu halten.

Wie in Abbildung 4.2 dargestellt, setzt sich die Architektur aus mehreren modularen, lose gekoppelten Komponenten zusammen. Diese Struktur ermöglicht eine hohe Flexibilität, erleichtert die Wartung und erlaubt einfache Anpassungen an unterschiedliche Einsatzszenarien.

**Skalierung und Hochverfügbarkeit** Artemis bietet umfangreiche Möglichkeiten zur Skalierung und Hochverfügbarkeit. Mehrere Serverinstanzen können zu Clustern zusammengeschaltet und bei Bedarf zu sogenannten Super-Clustern erweitert werden. Dadurch wird eine dynamische Lastverteilung erreicht und die Fehlertoleranz erhöht. Dank der Self-Healing-Funktionalität kann der Broker bei Konfigurations- oder Netzwerkfehlern automatisch Maßnahmen zur Wiederherstellung ergreifen.

Die Integration in bestehende Enterprise-Systeme wird durch standardisierte Schnittstellen wie Java Connector Architecture (JCA) erleichtert. Darüber hinaus existieren Bindings für zahlreiche Programmiersprachen, wodurch Artemis auch für heterogene Systemlandschaften attraktiv ist.

**Hardwareanforderung** Die Hardwareanforderungen von Apache ActiveMQ Artemis sind im Vergleich zu anderen Message-Brokern moderat, was Artemis insbesondere für den produktiven Einsatz in hochautomatisierten Produktionsumgebungen attraktiv macht. Grundsätzlich hängt der Ressourcenbedarf stark von der Konfiguration, der Anzahl gleichzeitiger Verbindungen sowie dem Nachrichtenvolumen ab. Dennoch lassen sich anhand von Benchmarks und der offiziellen Dokumentation allgemeine Aussagen treffen.

Im Leerlauf benötigt Artemis nur geringe CPU- und Speicherressourcen. Bereits mit einer Konfiguration von **2 vCPUs und 4 GB RAM** kann ein lauffähiges System betrieben werden. Für den stabilen Betrieb in produktiven Umgebungen empfiehlt die Apache Software Foundation jedoch eine Ausstattung von **mindestens 4–8 vCPUs und 8–16 GB RAM**, insbesondere bei Einsatz von Persistenzmechanismen und Clustering [3].

Bei Verwendung von Clustering oder Replikation steigen die Hardwareanforderungen entsprechend an, da zusätzliche Ressourcen für Synchronisation und Datenreplikation benötigt werden. Im Gegenzug verbessert sich dadurch jedoch die Ausfallsicherheit erheblich, was in produktionskritischen Szenarien einen entscheidenden Vorteil darstellt.

**Leistungsbewertung** In einer vergleichenden Leistungsstudie von Appel et al. (2023) zeigt sich, dass Artemis insbesondere in Umgebungen mit moderatem Durchsatz und niedriger Latenz überzeugende Ergebnisse liefert. Zwar erreicht es bei extrem hohen

Lasten nicht die Effizienz spezialisierter Event-Streaming-Plattformen, ist jedoch für klassische Messaging-Szenarien in produktionsnahen Systemen – wie sie auch in der hochautomatisierten Halbleiterfertigung auftreten – sehr gut geeignet [6].

**Vergleich mit ActiveMQ Classic** Tabelle 3 zeigt die wesentlichen Unterschiede zwischen ActiveMQ Classic und ActiveMQ Artemis in kompakter Form.

Tabelle 3: Vergleich ActiveMQ (Classic) und ActiveMQ Artemis [7]

| **Eigenschaft** | **ActiveMQ (Classic)** | **ActiveMQ Artemis** |
|---|---|---|
| Architektur | Traditionelle Broker-Architektur; begrenzte Skalierbarkeit bei hoher Last | Moderne, asynchrone Architektur; hohe Skalierbarkeit und Performance |
| Speicherverwaltung | Speicherintensiv; aufwändige Garbage Collection | Effizientes Paging; besseres Handling großer Nachrichten |
| Protokollunterstützung | JMS, AMQP, MQTT, STOMP | Gleiche Standards, moderner implementiert; höhere Performance bei AMQP/MQTT |
| Cluster-Fähigkeit | Veraltete Master-Slave-Replikation; Failover erfordert Neustarts | Shared-Nothing-Ansatz; hochverfügbar mit Live-Backup |
| Nachrichtenpersistenz | KahaDB als Standard; eingeschränkte I/O-Leistung bei Last | Journal-basiert; hohe Schreibgeschwindigkeit; asynchrones Speichern möglich |
| Verwaltung/Monitoring | JMX-basiert; begrenzte Web-UI | Erweiterte Web-Konsole; CLI-Tools und REST-Management |
| Nachrichtenrouting | Statisches Routing | Erweiterte Routing-Optionen; Adressen und Multicast |
| Skalierbarkeit | Architekturbedingt eingeschränkt; ungeeignet für Cloud-Umgebungen | Horizontal skalierbar; optimiert für Container/Kubernetes |
| Performance | Gut für kleine/mittlere Szenarien | Hoher Durchsatz; niedrige Latenz bei vielen Clients |

### 4.2.4 Vergleichende Übersicht der Systeme

Tabelle 4: Vergleich zentraler Eigenschaften von Apache ActiveMQ Artemis, NATS und MQTT 5.0

| **Eigenschaft** | Artemis | NATS | MQTT 5.0 |
|---|---|---|---|
| **Architektur** | Broker-basiert, Queue-orientiert, JMS-kompatibel | Leichtgewichtig, Cluster-orientiert | Broker-basiert, Topic-orientiert |
| **Nachrichtenprotokoll** | AMQP, STOMP, MQTT, OpenWire, Core | Eigenes Protokoll (leichtgewichtig), MQTT 3.1 | MQTT 5.0 (Binary Protokoll) |
| **Nachrichtenmodelle** | Pub/Sub, Queues, Request/Reply, Routing | Pub/Sub, Request/Reply, Echtzeitmodi | Nur Pub/Sub |
| **Persistenz** | Journalbasiert, Paging, Transaktionsfähig | Optional via JetStream | Vom Broker abhängig |
| **Skalierbarkeit** | Clustering, Replikation | Supercluster-Architektur, dynamisch skalierbar | Brokerabhängig, nicht nativ spezifiziert |
| **Client-Support** | Java nativ, andere via Protokolle | 30+ Sprachen (Go, Rust, Java, ...) | Weit verbreitet, viele Implementierungen |
| **QoS** | Transaktionen, Deduplizierung, Priorisierung | QoS via JetStream (at least once) | QoS 0, 1, 2 gemäß Standard |
| **Sicherheit** | JAAS, TLS, Authentifizierung | TLS, ACLs, Benutzerrollen | TLS, Authentifizierung, Reason Codes |
| **Management** | Web-Konsole, JMX, CLI | CLI, Monitoring-Tools, Self-healing | Brokerabhängig (z. B. Mosquitto UI, HiveMQ Admin) |
| **Besonderheiten** | Hohe Flexibilität, AMQP-kompatibel | Extrem geringe Latenz, Docker 3 MB, Fehlertoleranz | Eigenschaften über Properties steuerbar |

# 5 Material und Methoden

Zwar lassen sich viele Funktionalitäten und technische Eigenschaften der betrachteten Middleware-Lösungen aus der Dokumentation oder durch Marktanalysen erschließen, jedoch ist deren tatsächliche Leistungsfähigkeit unter produktionsnahen Bedingungen nur durch praktische Tests verlässlich zu bewerten.

Insbesondere für den geplanten Austausch des bestehenden TIBCO-Busses ist es essenziell, die Belastbarkeit, Skalierbarkeit, Latenz und Stabilität alternativer Systeme unter realitätsnaher Last zu untersuchen. Dazu wurden Testumgebungen entwickelt, die auf echten Produktionsdaten basieren und sowohl typische Kommunikationsmuster als auch Extrembelastungen simulieren. Diese Tests ermöglichen es, systematisch festzustellen, ob eine Middleware wie ActiveMQ Artemis die kritischen Anforderungen erfüllt – etwa in Bezug auf Nachrichtenfrequenz, Persistenzverhalten oder Reaktionszeit bei Ausfällen.

## 5.1 Testumgebung

Für die Durchführung weiterführender Tests wurde die integrierte Entwicklungsumgebung (IDE) Eclipse in der Version 2023-12 (4.30.0) verwendet. Zusätzlich stellte die SYSTEMA GmbH ein internes Client-Server-Framework (CSFW) zur Verfügung, das eine flexible und effiziente Konfiguration der Kommunikationsstrukturen zwischen Sendern und Empfängern erlaubt. Dieses Framework bildet die technische Grundlage für die Implementierung sämtlicher Testkomponenten und zeichnet sich durch folgende Besonderheiten aus:

- **Modularer Aufbau:** Die einzelnen Komponenten (z. B. Sender, Empfänger, Protokolladapter) sind als eigenständige Module konzipiert, wodurch einzelne Funktionalitäten bei Bedarf leicht ausgetauscht oder erweitert werden können.

- **Umfangreiche Konfigurationsmöglichkeiten:** Alle relevanten Einstellungen – wie Verbindungsparameter, Topic-Mappings oder Wiederholungsstrategien – werden in externen Konfigurationsdateien abgelegt.

- **Hohe Wiederverwendbarkeit:** Da Verhalten und Einsatzszenarien vollständig über Konfigurationsdateien gesteuert werden, kann das Framework in unterschiedlichen Projekten genutzt werden, ohne den Anwendungs- oder Testcode ändern zu müssen.

## 5.2 Speicherung von Nachrichten

Zur späteren Reproduktion der Nachrichten ist es erforderlich, diese während einer Aufzeichnung persistent zu speichern. Dabei wurde eine bewusste Abwägung zwischen dem in Kapitel 3.6 beschriebenen Nachrichtenformat *XML* und alternativen Speicherlösungen vorgenommen. In Betracht gezogen wurden insbesondere die Nutzung einer eingebetteten SQLite-Datenbank sowie der Einsatz externer Datenbanksysteme wie PostgreSQL oder MongoDB.

Da ein vollständiges Datenbanksystem für den gegebenen Anwendungsfall als überdimensioniert erscheint, wurde die Betrachtung auf **SQLite** als leichtgewichtige Alternative zu XML-Dateien eingegrenzt.

### 5.2.1 SQLite

SQLite ist eine weit verbreitete, leichtgewichtige, serverlose SQL-Datenbank, die vollständig in eine Anwendung eingebettet werden kann. Sie speichert sämtliche Daten in einer einzigen Datei und unterstützt ACID-konforme Transaktionen sowie strukturierte Abfragen mittels SQL.

#### Vorteile

- **Strukturierte Datenspeicherung:** Daten lassen sich effizient anhand bestimmter Attribute durchsuchen, filtern und analysieren.
- **Transaktionssicherheit:** Schreibvorgänge erfolgen atomar, konsistent, isoliert und dauerhaft.
- **Portabilität:** Die Datenbank ist plattformunabhängig und lässt sich ohne Konvertierung weiterverwenden.

#### Nachteile

- **Erhöhter Implementierungsaufwand:** Der Einsatz erfordert das Anlegen und Pflegen eines Datenbankschemas sowie die Verwaltung von Verbindungslogik und SQL-Abfragen.
- **Geringe Transparenz:** Eine unmittelbare Sicht auf gespeicherte Daten ist nur über Zusatztools oder Programme möglich.

- **Technologische Abhängigkeiten:** Der Einsatz externer Bibliotheken und erweiterte Fehlerbehandlung sind erforderlich.

### 5.2.2 Begründung der Formatwahl

Da die zu speichernden Nachrichten in ihrem ursprünglichen XML-Format vorliegen und lediglich sequenziell sowie in korrekter Reihenfolge archiviert werden sollen, ist der Einsatz einer relationalen Datenbank derzeit nicht erforderlich. Weder komplexe Datenabfragen noch Mehrbenutzerzugriffe oder Integritätsprüfungen sind Teil der Anforderungen. Ein Dateispeicher ermöglicht in diesem Fall eine unkomplizierte und transparente Speicherung bei gleichzeitig minimalem Implementierungsaufwand.

Die Verwendung von XML als Speicherformat ermöglicht darüber hinaus eine direkte Sicht auf die Daten und erleichtert einfache Validierung und manuelle Analyse. Die Entscheidung für ein dateibasiertes Vorgehen trägt somit zur Reduzierung von Systemkomplexität und Wartungsaufwand bei.

Für zukünftige Erweiterungen – etwa bei der Anbindung weiterer Abteilungen oder bei wachsendem Bedarf an komplexen Abfragen – bleibt die Option eines Umstiegs auf SQLite oder ein vollwertiges Datenbanksystem jedoch offen. Aufgrund der klaren Trennung von Datenhaltung und Verarbeitungslogik wäre eine nachträgliche Migration mit überschaubarem Aufwand realisierbar.

## 5.3 Nachrichtenaufzeichnung und Reproduktion

Zur Erzeugung realitätsnaher Testergebnisse wurde auf produktive Nachrichten aus dem laufenden Fertigungsbetrieb zurückgegriffen. Zu diesem Zweck wurde eine Listener-Komponente (*ListenerSaver*) entwickelt, welche eintreffende Nachrichten aufzeichnet und in strukturierter Form in einer XML Datei persistiert.
Auf Basis dieser Daten ermöglicht eine korrespondierende Sender-Komponente (*EventSender*) die Reproduktion der Kommunikationssequenzen. Dabei werden die zuvor aufgezeichneten Nachrichten in identischer Reihenfolge und mit dem originalen zeitlichen Abstand erneut versendet.

### 5.3.1 Implementierung des Nachrichtenspeichers: ListenerSaver (Listing 4)

Die Klasse *ListenerSaver* übernimmt die Aufgabe, eintreffende Bus-Nachrichten asynchron und thread-sicher in eine XML-Datei zu speichern. Sie ist als Serverdienst konzipiert und

erweitert die abstrakte Basisklasse *ASysListenService* aus dem Systema-Framework. Ziel dieser Komponente ist es, eine lückenlose Protokollierung aller empfangenen Nachrichten zu gewährleisten, ohne die Echtzeitfähigkeit des Systems zu beeinträchtigen.

**Initialisierung und Konfiguration**
Die Methode `init(...)` wird beim Start des Dienstes aufgerufen und liest die Konfiguration aus dem Systema-Konfigurationssystem ein. Dabei wird insbesondere der Pfad zur Zieldatei (standardmäßig `C:\temp\bus-messages.xml`) geladen.
Zusätzlich wird über eine weitere Methode geprüft, ob die Datei bereits initialisiert wurde. Falls nicht, wird sie mit einem leeren XML-Root-Tag (`<root></root >`) vorbereitet. Dies ist notwendig, da XML-Dokumente strukturell ein einzelnes Wurzelelement (Root-Element) erfordern, welches sämtliche untergeordneten Elemente umschließt. Wird diese Vorgabe nicht eingehalten, führt dies beim späteren Einfügen von Nachrichten zu einem formal ungültigen XML-Dokument, was sowohl das Parsen als auch die Weiterverarbeitung erheblich erschwert oder unmöglich macht. Durch das Voranlegen eines Root-Tags wird sichergestellt, dass alle eingehenden Nachrichten korrekt innerhalb dieses übergeordneten Elements eingefügt werden können, wodurch die Konsistenz und Validität der gesamten Datei gewahrt bleibt.

**Ereignisbehandlung**
Bei jedem eingehenden Ereignis wird die Methode `handleEventNow(...)` aufgerufen. In ihr wird der Zeitpunkt der Nachricht mittels Erstellen eines Zeitstempels im Format yyyy-MM-dd HH:mm:ss.SSS festgehalten und anschließend die Methode `saveMessageToQueue(...)` aufgerufen. Diese Methode übernimmt das Einreihen der Nachricht in eine Blocking Queue, wobei sie zuvor mit dem Zeitstempel sowie dem Betreff (`subject`) ergänzt und formatiert wird.

**Nachrichtenformatierung**
Die Formatierung der Nachricht erfolgt in der Methode `formatMessage(...)`. Sie stellt sicher, dass das XML korrekt aufgebaut ist, ergänzt den `timestamp` als Attribut des `<message>`-Tags und fügt den Betreff als zusätzliches XML-Element <subject>ein. Damit bleibt die Datei sowohl maschinenlesbar als auch strukturiert durchsuchbar.

**Asynchrone Verarbeitung**
Die Verarbeitung eingehender Nachrichten erfolgt asynchron über einen dedizierten Hintergrund-Thread, der beim Initialisieren des Systems mittels `startQueueProcessor()` gestartet wird. Dieser entnimmt fortlaufend Nachrichten aus der thread-sicheren Warteschlange `MESSAGE_QUEUE` und übergibt diese an die Methode `appendMessageToFile(...)` zur Speicherung weiter.
Durch diese Entkopplung von Nachrichteneingang und Dateioperationen wird sichergestellt,

dass blockierende I/O-Zugriffe (z. B. Dateischreibvorgänge) den kontinuierlichen Empfang nicht beeinträchtigen.

Die Verarbeitung der Queue erfolgt mittels eines `poll()`-Mechanismus mit Timeout. Dies erlaubt eine effiziente Behandlung von Leerlaufphasen und verhindert gleichzeitig, dass der Thread dauerhaft blockiert oder in einen potenziellen Deadlock-Zustand übergeht.

**Dateizugriff und Synchronisierung**
Beim Schreiben in die Datei wird sichergestellt, dass das XML-Dokument stets wohlgeformt und gültig bleibt. Hierzu wird der aktuelle Inhalt der Datei bis zum schließenden `</root>`-Tag gelesen, dieses temporär entfernt, die neue Nachricht eingefügt und anschließend das Tag erneut hinzugefügt.

Der gesamte Zugriff auf die Datei ist über ein synchronisiertes Objekt `FILE_LOCK` geschützt. Dadurch werden konkurrierende Schreibzugriffe verhindert und sichergestellt, dass die Konsistenz und Integrität der XML-Struktur auch bei parallelen Verarbeitungsvorgängen gewahrt bleibt.

Für das Einfügen der Nachrichten wird gezielt an das Dateiende vor das schließende Root-Tag navigiert und der neue Nachrichtentext davor eingefügt mit gleichzeitigem Überschreiben des Tags. Anschließend wird das Root-Tag wieder ergänzt. Auf diese Weise bleibt die XML-Struktur erhalten, und alle Nachrichten sind syntaktisch korrekt innerhalb des Root-Elements eingebettet.

### 5.3.2 Beschreibung und Funktionsweise der Klasse EventSender (Listing 5)

Die Klasse *EventSender* dient zur zeitlich gesteuerten Wiederholung (Rekonstruktion) zuvor aufgezeichneter Bus-Nachrichten in einem verteilten Nachrichtensystem. Sie erweitert die abstrakte Basisklasse *ASysListenService* und implementiert somit einen Dienst innerhalb einer serverbasierten Architektur, die auf der Middleware von Systema basiert.

**Initialisierung und Konfiguration** `init(ISysServer server, ISysConfigItem config)` wird beim Start des Dienstes aufgerufen. Dabei wird der Pfad zu einer XML-Datei geladen, die zuvor aufgezeichnete Nachrichten enthält. Der Pfad wird entweder aus der Konfigurationsdatei übernommen oder auf den Standardwert `C:\temp\bus-messages.xml` zurückgesetzt, sofern keine Konfiguration vorliegt. Die Konfiguration erfolgt über den Systema-Konfigurationsmechanismus mittels `getMergedConfiguration()`.

**Ereignisverarbeitung** `handleEventNow(...)` wird durch das Middleware-System aufgerufen, sobald ein Ereignis detektiert wird. Im gegebenen Fall dient sie als Trigger für die Methode `replayMessages()`, welche den Prozess der Nachrichtensimulation asynchron startet. Die Nutzung von `CompletableFuture.runAsync()` in Kombination mit einem statischen ExecutorService (`EXECUTOR`) erlaubt eine nicht-blockierende und parallele Ausführung, was der Skalierbarkeit des Systems zugutekommt.

**Einlesen und Parsen der Nachrichten** `readMessagesFromFile()` liest die XML-Datei zeilenweise ein. Dabei werden mittels regulärer Ausdrücke (Pattern) gezielt bestimmte Informationen extrahiert:

- Zeitstempel (`timestamp`) im Format 'yyyy-MM-dd HH:mm:ss.SSS'
- Thema bzw. Adressat der Nachricht (<`subject`>)

Die Nachrichteninhalte selbst werden als XML-Fragment in einem *StringBuilder* gesammelt. Nach vollständiger Verarbeitung werden sie in Objekte der inneren Hilfsklasse *MessageData* überführt, die neben dem Nachrichteninhalt auch Zeitstempel und Adressat speichert. Dies ermöglicht eine saubere Kapselung und Trennung der Daten.

**Simulation der Nachrichtenaussendung** `simulateMessageReplay(...)` sortiert die Nachrichten chronologisch nach ihrem Zeitstempel und berechnet auf dieser Basis die relativen Verzögerungszeiten (Delays) zwischen den Nachrichten. Die tatsächliche Wiedergabe erfolgt zeitversetzt über `EXECUTOR.schedule(...)`, wobei ein Multiplikator (`timemanipulation`) verwendet werden kann, um die Wiedergabegeschwindigkeit zu beeinflussen. Standardmäßig ist dieser auf 1 gesetzt, d.h., die Nachrichten werden in Echtzeit gesendet.

**Aussenden der Nachrichten** `sendMessage(...)` ist verantwortlich für das Senden der Nachricht über den Bus. Dabei wird eine neue `ISysMessageItem`-Instanz erstellt und mit Metadaten angereichert (z.B. Zeitstempel, Zähler). Die Nachricht wird anschließend über ein dynamisch erzeugtes `ISysSubject` an das Bus-System veröffentlicht.

Ein Mechanismus zur Duplikatsvermeidung ist integriert: Nur Nachrichten mit einem neuen Zeitstempel (in curmeskey) werden gesendet. Dadurch wird verhindert, dass identische Nachrichten mehrfach publiziert werden, sofern sie im Eingangsdatensatz mehrfach vorhanden sind.

**Architektur- und Designaspekte**

- **Nebenläufigkeit**: Durch den Einsatz eines *ScheduledExecutorService* wird eine hohe Kontrolle über die Parallelität und die Ausführungszeitpunkte erreicht. Dies ist insbesondere bei zeitkritischen Systemen essentiell.
- **Kapselung**: Die Hilfsklasse *MessageData* kapselt die Struktur einer Nachricht vollständig und bietet eine saubere API.
- **Trennung von Anliegen** (Separation of Concerns): Die einzelnen Funktionen der Klasse – Dateieinlesen, Nachrichtenparsing, Zeitsteuerung, Nachrichtensendung – sind modular getrennt implementiert.
- **Fehlertoleranz**: Fehler in der Verarbeitung werden über `try-catch-Blöcke` abgefangen und protokolliert. Eine Verbesserung wäre hier eine zentralisierte Fehlerbehandlung.

## 5.4 Belastungstest und Latenzmessung

Für weiterführende Analysen hinsichtlich der Latenzzeiten und Belastbarkeit des Systems wurde ein ergänzender Testansatz implementiert. Hierbei generiert eine separate Sender-Instanz identische Nachrichten mit gleichbleibender Frequenz. Ein synchron arbeitender Empfänger nimmt diese Nachrichten entgegen und protokolliert neben den Empfangszeitpunkten auch etwaige Abweichungen, wie z.B. Nachrichtenverluste oder Latenzspitzen. Die gewonnenen Daten dienen der quantitativen Bewertung der Leistungsfähigkeit der eingesetzten MOM unter praxisnahen Bedingungen.

### 5.4.1 Konstantes Senden von Nachrichten

Zur kontinuierlichen Ermittlung der Latenz im Nachrichtensystem wurde ein periodisch arbeitender Nachrichtengenerator implementiert. Ziel dieser Komponente ist es, systematisch und in festen Zeitintervallen strukturierte Testnachrichten an mehrere Empfänger zu senden. Dies erlaubt eine spätere Analyse der Übertragungsdauer zwischen Versand- und Empfangszeitpunkt.

```
1 /**
2  * The MesSender service periodically creates and sends structured
      messages to multiple event senders.
3  * Each message includes a timestamp and an incrementing counter
      value.
```

```
4  * This class is useful for performance testing, throughput analysis,
       or latency measurement setups.
5  */
6 public class MesSender extends ASysServerService {
7
8   private static final long serialVersionUID = 1123568;
9
10  /** Scheduler used for periodic message sending. */
11  private final ScheduledExecutorService scheduler = Executors.
       newSingleThreadScheduledExecutor();
12
13  /** Formatter used for timestamp formatting (HH:mm:ss.SSS). */
14  private final DateTimeFormatter DATE_FORMAT = DateTimeFormatter.
       ofPattern("HH:mm:ss.SSS");
15
16  /** Atomic counter used to track how many messages have been sent.
       */
17  public static AtomicInteger count = new AtomicInteger(0);
18
19  /** Delay between messages in milliseconds (default = 50ms). */
20  public static int sendDelay = 50;
21
22  /** The message template used for publishing. */
23  ISysMessage message = FSysMessage.createMessage();
24
25  /** Inner message structure holding the contents of the message. */
26  ISysMessageItem contents = message.addItem("CONTENTS", FSysMessage.
       createMessage());
27  /**
28   * Initializes the service by preparing the message structure and
        scheduling
29   * the periodic sending task.
30   *
31   * @param server the system server
32   * @param configItem the configuration for the service
33   * @throws ASysException if initialization fails
34   */
35  @Override
36  public void init(ISysServer server, ISysConfigItem configItem)
       throws ASysException {
37    super.init(server, configItem);
38    contents.addItem("Message", "abc123");
39    scheduler.scheduleWithFixedDelay(this::publishEvent, 500,
         sendDelay, TimeUnit.MILLISECONDS);
40  }
41
42  /**
43   * Not used in this implementation.
44   *
45   * @param msg unused message parameter
46   * @return always returns null
47   * @throws ASysException unused
48   */
49  @Override
50  protected ISysMessage runNow(ISysMessage msg) throws ASysException
       {
51    return null;
52  }
53
54  /**
55   * Prepares and sends a new message to all available event senders.
```

```
56     * Each message includes the current counter and timestamp.
57     * Messages are sent to all event channels named "Event0" through "
           Event9".
58     * If a sender is unavailable, a message is printed to the console.
59     */
60    protected void publishEvent() {
61      try {
62        contents.clear();
63        contents.addItem("COUNT", count.getAndIncrement());
64
65        String timestamp = LocalDateTime.now().format(DATE_FORMAT);
66        contents.addItem("timestamp", timestamp);
67
68        for (int i = 0; i < 10; i++) {
69          ISysEventSender sender = getEventSender("Event" + i);
70          if (sender != null) {
71            sender.send(null, message);
72          } else {
73            System.out.println("no Sender");
74          }
75        }
76      } catch (Throwable th) {
77        th.printStackTrace(); // consider using a proper logging
              mechanism
78      }
79    }
80  }
```

Listing 1: Konstanter Nachrichtenversand zur Latenzmessung

Technisch basiert die Lösung auf einem `ScheduledExecutorService`, welcher die Methode `publishEvent()` in einem konstanten Zeitabstand (`sendDelay`) ausführt. Diese Methode erzeugt für jede Periode eine neue Nachricht vom Typ `ISysMessage`, ergänzt um relevante Informationen:

- **Message:** Ein statischer Text zur Identifikation (z. B. `abc123`)
- **COUNT:** Ein inkrementeller Zählerwert zur eindeutigen Referenzierung
- **timestamp:** Die aktuelle Uhrzeit im Format `HH:mm:ss.SSS` zur späteren Latenzbestimmung

Die Nachrichten werden anschließend an mehrere `ISysEventSender`-Instanzen verteilt. In der aktuellen Implementierung erfolgt der Versand über zwei benannte Sender (z. B. `Event0` und `Event1`), wobei jede Nachricht mit jedem Sender an jeweils ein anderes Topic gesendet wird. Dadurch entsteht eine erhöhte Last, die für laufzeitkritische Tests relevant ist.

Die gewählte Struktur stellt sicher, dass die Nachrichten in definierten Abständen erzeugt und gesendet werden. Durch die exakte Zeitstempelung und Zählung lassen sich die

empfangenen Nachrichten später eindeutig zuordnen und hinsichtlich der Verzögerung analysieren. Der Aufbau eignet sich somit hervorragend für Performanz- und Belastungstests im Rahmen der Systembewertung.

### 5.4.2 Empfangen und Latenzberechnung der gesendeten Nachrichten

Zur Ermittlung der Latenzzeit und der Zuverlässigkeit der Nachrichtenübertragung wird im Empfänger-Modul der kontinuierlich gesendete Nachrichtenstrom ausgewertet. Dabei werden neben dem Empfangszeitpunkt auch der Zählerstand der Nachrichten analysiert, um verlorene Nachrichten und ungewöhnlich lange Verzögerungen zu erkennen.

```
/**
 * ReceiverClient1 is a listener service that receives messages and verifies
 * their timing and order. It calculates the time between consecutive messages and logs
 * errors if messages arrive too late or if message loss is detected.
 */
public class ReceiverClient1 extends ASysListenService {

  private static final long serialVersionUID = 1122334455L;

  /** Formatter for parsing timestamps from received messages. */
  private static final DateTimeFormatter TIME_FORMATTER =
      DateTimeFormatter.ofPattern("HH:mm:ss.SSS");

  private LocalTime lastTimestamp = null;
  private int lastCounter = 0;

  /** Maximum delay in milliseconds allowed between two messages. */
  private final int maxAcceptableDelay = MesSender.sendDelay + 100;

  /**
   * Handles an incoming message by validating its timestamp and message order.
   * Logs errors when a message is late or when messages appear to be missing.
   *
   * @param message the received message
   * @param subject the subject (topic) of the message
   * @throws ASysException if message processing fails
   */
  @Override
  protected void handleEventNow(ISysMessage message, ISysSubject subject) throws ASysException {
    String countStr = message.getItem("CONTENTS").getItemX("COUNT").getValue().toString();
    String timestampStr = message.getItem("CONTENTS").getItemX("timestamp").getValue().toString();

    int currentCounter = Integer.parseInt(countStr);
    LocalTime currentTimestamp = LocalTime.parse(timestampStr, TIME_FORMATTER);

    if (lastTimestamp != null && lastCounter != 0) {
```

```
36        Duration diff = Duration.between(lastTimestamp,
              currentTimestamp);
37        long millis = diff.toMillis();
38
39        int lostMessages = currentCounter - lastCounter;
40
41        if (millis > maxAcceptableDelay) {
42          CSysLog.log(ISysConstants.COMPONENT_TYPE,
43              "Timeout: " + millis + " ms between messages. Count: " +
                  currentCounter,
44              ISysLogLevels.ERROR, this);
45          ErrorHandling(millis + currentCounter, message);
46        }
47
48        if (lostMessages > 0) {
49          CSysLog.log(ISysConstants.COMPONENT_TYPE,
50              "Lost messages detected: " + lostMessages,
51              ISysLogLevels.ERROR, this);
52        }
53      }
54
55      lastCounter = currentCounter + 1;
56      lastTimestamp = currentTimestamp;
57    }
58
59    /**
60     * Handles timing errors by writing an entry to a local log file
            for later analysis.
61     *
62     * @param millis the delay in milliseconds + message counter
63     * @param message the message that caused the timing violation
64     */
65    private void ErrorHandling(long millis, ISysMessage message) {
66      String filename = "C:\\temp\\log.txt";
67
68      String timestamp = LocalDateTime.now().format(DateTimeFormatter.
            ofPattern("yyyy-MM-dd HH:mm:ss"));
69      String line = String.format("[%s] %s , %s", timestamp, millis,
            message);
70
71      try (BufferedWriter writer = new BufferedWriter(new FileWriter(
            filename, true))) {
72        writer.write(line);
73        writer.newLine();
74      } catch (IOException e) {
75        e.printStackTrace(); // Ideally use a logger
76      }
77    }
78  }
```

Listing 2: Empfangen und Latenzberechnung

Die Klasse `ReceiverClient1` erbt von der Basisklasse `ASysListenService` und überschreibt die Methode `handleEventNow`, welche bei Eintreffen einer Nachricht aufgerufen wird. Die eingehenden Nachrichten enthalten in ihrem Payload unter dem Knoten `CONTENTS` die Attribute `COUNT` (einen fortlaufenden Zähler) und `timestamp` (den Zeitstempel im Format `HH:mm:ss.SSS`).

Beim Empfang einer Nachricht werden diese Werte extrahiert und in entsprechende lokale Variablen eingelesen. Anschließend wird der aktuelle Zeitstempel mit dem Zeitstempel der vorherigen Nachricht verglichen, indem die Zeitdifferenz in Millisekunden berechnet wird. Diese Differenz wird mit einem definierten Schwellenwert (dem Sendeintervall plus einem Toleranzpuffer) verglichen. Überschreitet die Differenz diesen Schwellenwert, so wird ein Warnhinweis geloggt, da dies auf eine verzögerte oder unterbrochene Nachrichtenübertragung hindeuten kann. Gleichzeitig wird die Methode `ErrorHandling` anufgerufen. Diese erzeugt in einer Log Datei einen Eintrag mit genauer Zeitangabe, Zeitverzug der Nachricht und der Nachricht selbst, welche die Sendezeit und den Counter enthält. Dadurch kann trotz der hohen Sendefrequenz eine Auswertung realisiert werden.

Zusätzlich wird der Nachrichten-Counter verglichen, um festzustellen, ob eine oder mehrere Nachrichten verloren gegangen sind. Hierzu wird die Differenz der aktuellen und vorherigen Zählerwerte plus eins. Sind Nachrichten verloren gegangen, wird dies als kritischer Fehler (ERROR Log-Level) protokolliert, da eine zuverlässige Nachrichtenübermittlung essentiell für das System ist.

Diese Vorgehensweise erlaubt es, sowohl zeitliche Latenzen als auch Datenverluste im Nachrichtentransport frühzeitig zu erkennen und entsprechend zu protokollieren, was die Grundlage für eine robuste Analyse der Kommunikationsqualität bildet.

Der Einsatz von Java-Standardklassen wie `LocalTime` und `Duration` für die Zeitberechnung gewährleistet dabei eine präzise und effiziente Verarbeitung der Zeitstempel, während die Verwendung von differenzierten Log-Levels eine gezielte Fehlerbehandlung und Überwachung ermöglicht.

# 6 Durchführung und Auswertung der Tests

Dieser Abschnitt befasst sich mit der praktischen Durchführung sowie der Auswertung der zuvor konzipierten Lasttests. Grundlage der Bewertung bilden die aus der Analysephase gewonnenen Lastkennzahlen (vgl. Abschnitt 3.1).

Ziel ist es, die Belastbarkeit des implementierten Nachrichtensystems unter realistischen sowie unter maximierten Bedingungen zu überprüfen. Hierzu wurden zwei unterschiedliche Testszenarien entworfen:

- **Test 1: Dauerlast** – Simulation des typischen Nachrichtentraffics inklusive Lastspitzen gemäß der Produktionsdatenanalyse.

  Im diesem Szenario wurde ein dauerhafter Betrieb des Systems über mehrere Minuten simuliert. Um die Produktionslast realitätsnah abzubilden, wurde der Timer des verwendeten Schedulers auf 18 Millisekunden konfiguriert. Zusätzlich wurden fünf weitere Senderinstanzen aktiviert, um die Anzahl der pro Tick erzeugten Nachrichten zu erhöhen.

  Durch diese Konfiguration entsteht eine kontinuierliche Speicherlast von etwa 350 KB/s. Dies entspricht ungefähr dem Vierfachen der in der Analyse gemessenen durchschnittlichen Produktionslast.

- **Test 2: Maximallast** – Belastung des Systems mit maximal möglichem Nachrichtenvolumen zur Bestimmung der Obergrenze.

  Zur Bestimmung der maximalen Leistungsfähigkeit des Systems wurde im zweiten Testszenario die Erzeugung von Nachrichten gezielt intensiviert. Grundsätzlich kann dies auf zwei Wegen erreicht werden: durch Erhöhung der Sendefrequenz (d.h. Verkürzung des Intervalls im Scheduler) oder durch die gleichzeitige Nutzung einer größeren Anzahl von Topics. Im Rahmen dieses Tests wurde ein kombinierter Ansatz verfolgt: Die Zahl der verwendeten Topics wurde verdoppelt und das Scheduler-Intervall auf 10 ms reduziert.

## 6.1 TIBCO EMS

### 6.1.1 Test 1: Dauerlast

Während der Durchführung des Lasttests konnte über das firmeneigene Monitoring-Tool der SYSTEMA GmbH ein wiederkehrendes Muster im Nachrichtentransfer beobachtet werden: In regelmäßigen Intervallen kam es zu kurzen Unterbrechungen der Datenübertragung, gefolgt von einem plötzlichen Nachsenden der zuvor zurückgehaltenen Nachrichten. Diese Aussetzer betrugen jeweils etwa 3000 bis 4000 Millisekunden. Abbildung 6.1 veranschaulicht dieses Verhalten anhand der Messung der übertragenen Nachrichten pro Sekunde.

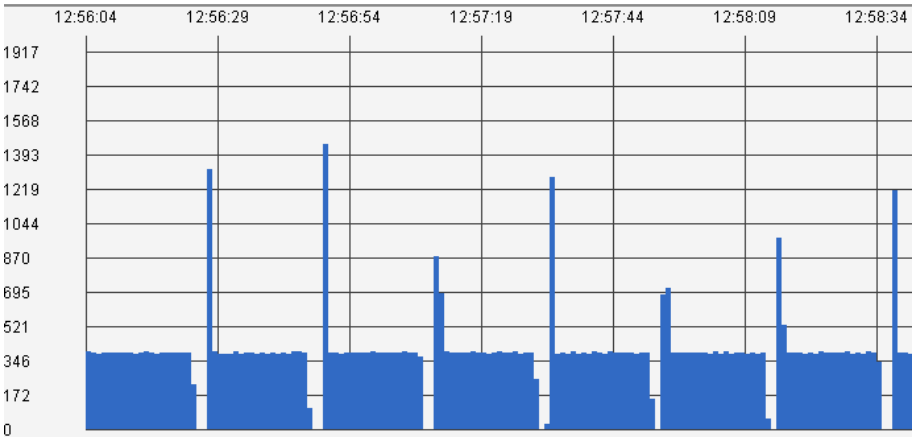



Abbildung 6.1: Nachrichten/s mit erkennbaren Übertragungslücken

**Systematische Eingrenzung der Ursache:** Zur Identifikation der Fehlerursache wurden verschiedene Hypothesen getestet. Zunächst wurde überprüft, ob andere Komponenten des Systems Einfluss auf das Verhalten haben könnten. Hierzu wurden sämtliche nicht benötigten Klassen aus dem Build entfernt und lediglich der relevante `MesSender` aktiviert. Diese Maßnahme führte jedoch zu keiner Veränderung des Verhaltens.

Im nächsten Schritt wurde untersucht, ob die Aussetzer zeitlich periodisch oder in Abhängigkeit von der Nachrichtenanzahl auftreten. Hierfür wurde die Frequenz der Nachrichtenaussendung durch Anpassen des Scheduler-Timers verändert. Das Ergebnis in Listing 6.2 zeigt, dass der Aussetzer nicht in festen zeitlichen Intervallen, sondern nach einer konstanten Anzahl versendeter Nachrichten auftrat – unabhängig von der absoluten Zeit. Dies deutet auf eine kapazitäts- oder speicherbezogene Ursache hin.

Weitere Tests mit variierter Nachrichtenfrequenz bestätigten diese Beobachtung: Unabhängig davon, ob Nachrichten in dichter oder entfernter Taktung gesendet wurden,

traten die Übertragungsunterbrechungen stets nach einer ähnlichen Anzahl von Nachrichten auf. Die Hypothese eines Verarbeitungsstaus konnte damit ausgeschlossen werden.

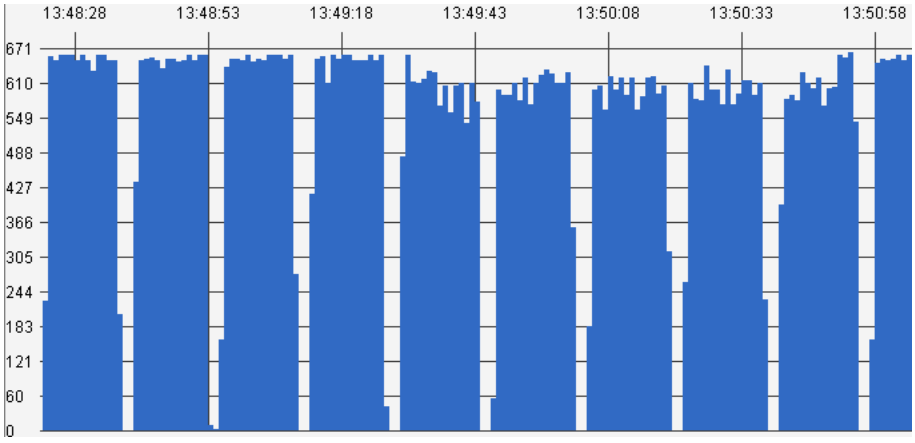



Abbildung 6.2: Aussetzer Bei hoher Last

**Untersuchung der Nachrichtengröße und Logging-Verhalten:** Anschließend wurde geprüft, ob die Größe der übertragenen Nachrichten das Verhalten beeinflusst. Mit Hilfe des `ReceiverClient` wurden Zeitstempel und Gesamtzahl der empfangen Nachrichten bei einem Aussetzer in einer Logdatei festgehalten. Wie in Abbildung 3 zu erkennen ist, verschiebt sich die Position der Aussetzer bei kleineren Nachrichten in spätere Testphasen, was auf eine proportionale Beziehung zur Gesamtmenge der übertragenen Daten schließen lässt.

Listing 3: Auszug aus der Logdatei mit dokumentierten Aussetzern

```
[2025-06-17 10:38:22] 4440
[2025-06-17 10:39:10] 4440

[2025-06-17 10:41:27] 4700
[2025-06-17 10:42:18] 4687

[2025-06-17 10:48:23] 4463
[2025-06-17 10:49:12] 4492
[2025-06-17 10:49:59] 4439

[2025-06-17 10:57:02] 4978
[2025-06-17 10:59:01] 4967

[2025-06-17 11:00:38] 4783
[2025-06-17 11:01:26] 4767
[2025-06-17 11:06:10] 4763
```

Schließlich wurde im Rahmen einer detaillierten Konsolenanalyse eine vorher unentdeckte Fehlermeldung identifiziert:

> Cannot rename file lukas-spiel-csfw-i1.log to lukas-spiel-csfw-i1_000.log in path ...

Diese Meldung weist auf ein Problem im internen Log-Rotationsmechanismus des Client-Server-Frameworks hin. Aufgrund einer fehlerhaften Namensvergabe konnte die Logdatei nicht mehr rotiert werden, da die verfügbaren Dateinamen (z. B. von `000` bis `999`) im Zielordner erschöpft waren. Der Versuch, die bestehende Logdatei umzubenennen, schlug fehl und führte offenbar zu einer temporären Blockade der IO-Ressourcen, wodurch die Nachrichtenverarbeitung für mehrere Sekunden unterbrochen wurde.

**Lösungsmaßnahme:** Zur Behebung dieses Problems wurde der gesamte Log-Ordner manuell gelöscht. Beim darauffolgenden Programmstart wurde ein neuer Ordner automatisch erzeugt, wodurch die Datei-Rotation wieder ordnungsgemäß funktionierte.
Wie in Abbildung 6.3 zu sehen konnten in nachfolgenden Testläufen keine weiteren Übertragungsaussetzer festgestellt werden.

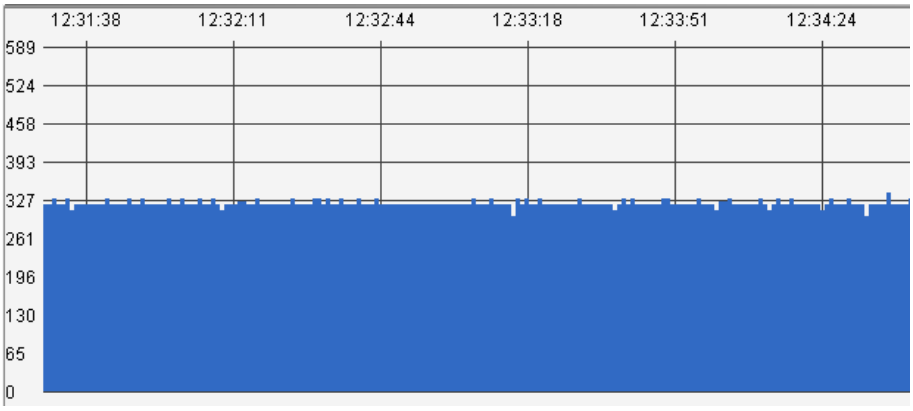



Abbildung 6.3: Nachrichten ohne Aussetzer

### 6.1.2 Test 2: Maximallast

Durch die veränderte Konfiguration konnte die Nachrichtenrate im Vergleich zu Test 1 nahezu verdoppelt werden. Das System wurde damit einer deutlich höheren Gesamtlast ausgesetzt, wodurch eine realistische Einschätzung der maximalen Verarbeitungskapazität ermöglicht wurde.

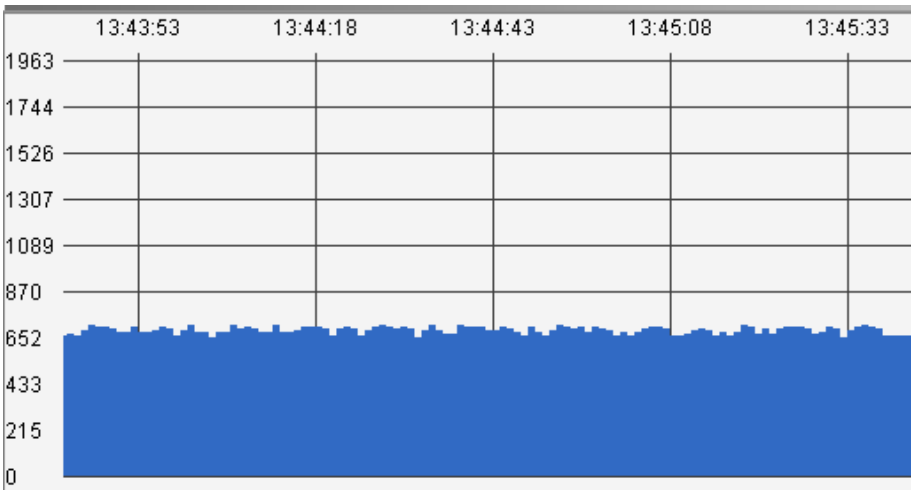



Abbildung 6.4: Nachrichten/s Hohe Last

## 6.2 ActiveMQ Artemis

Da sich die Konfiguration von ActiveMQ Artemis im XFab-Systemkontext zum Zeitpunkt der Tests noch in der Entwicklungsphase befindet, sind die folgenden Ergebnisse als vorläufige Indikation der Leistungsfähigkeit zu verstehen. Sie liefern jedoch bereits wertvolle Hinweise auf das Verhalten des Systems unter produktionsnaher Dauerbelastung.

In der aktuellen Konfiguration wird der ActiveMQ Artemis Broker als Cluster innerhalb einer Kubernetes-Umgebung betrieben. Das Cluster besteht aus insgesamt drei Brokern, die unterschiedliche Rollen einnehmen: Der erste Broker fungiert als primärer Knoten, während der zweite eine Replikation des primären Brokers darstellt um synchrone Persistenz zu gewährleisten. Der dritte Broker agiert als passiver Beobachter und übernimmt ausschließlich im Fehlerfall aktiv Aufgaben. Er überwacht kontinuierlich den Systemzustand und wird bei Eintritt einer Failover-Situation automatisch aktiviert. In diesem Fall übernimmt er die Rolle des Backup-Brokers, während der bisherige Replikationsknoten zum neuen primären Broker wird.

Zur Sicherstellung der Datenpersistenz speichert jeder Broker sämtliche eingehenden Nachrichten auf einem persistenten Netzwerkdateisystem (NFS). Dabei besitzt jeder Knoten ein eigenes dediziertes NFS-Share, um Datenverlust und Konflikte im Failover-Fall zu vermeiden.
Die Kommunikation mit den Clients erfolgt über einen vorgelagerten Loadbalancer, der zur Zeit lediglich der Ausfallsicherheit bei Wartungsarbeiten gilt. Werden im weiteren Entwicklungsverlauf mehrere Primäre Knoten hinzugefügt, wird dieser auch für eine gleichmäßige Verteilung der Verbindungen auf die verfügbaren Broker sorgen und somit zur Skalierbarkeit und Fehlertoleranz des Systems beitragen.

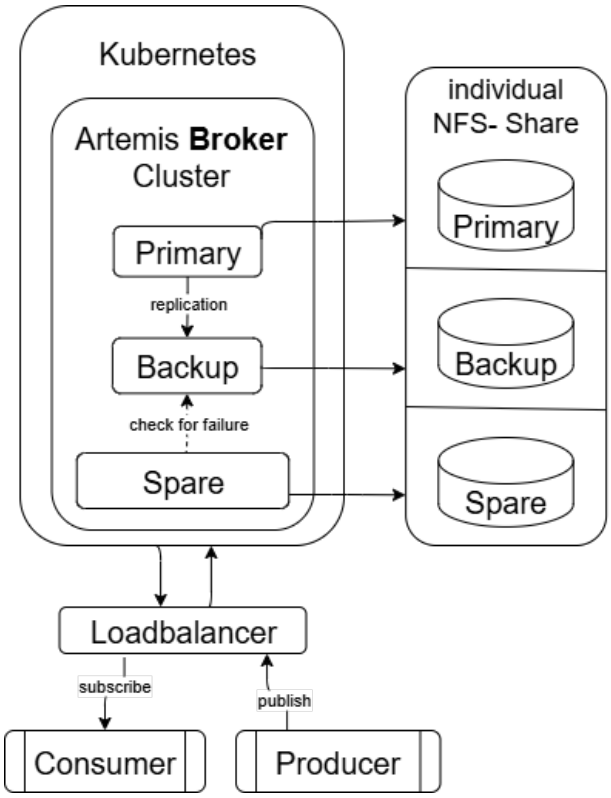



Abbildung 6.5: Artemis Struktur

### 6.2.1 Test 1: Dauerlast

Ein zentrales Merkmal der aktuellen Konfiguration ist die aktivierte Persistenzfunktion, welche der Erhöhung der Ausfallsicherheit dient. Alle eingehenden Nachrichten werden dabei vor der Weiterleitung zwischengespeichert.
Dieser Mechanismus führt zu aktuellem Zeitpunkt jedoch zu einer deutlich unruhigeren Verlaufskurve der übertragenen Nachrichten pro Sekunde, wie in Abbildung 6.6 ersichtlich. Es zeigen sich ausgeprägte Schwankungen sowie einzelne Lastspitzen, was auf den internen Verarbeitungsrhythmus der Queue- und Persistenzmechanismen zurückzuführen ist. Trotz dieser Schwankungen ist jedoch zu jedem Zeitpunkt gewährleistet, dass der momentane Standarddurchsatz von 65 Nachrichten pro Sekunde erreicht wird.

Positiv hervorzuheben ist, dass durch die Persistenz keine Nachrichten verloren gehen, auch bei kurzzeitigen Engpässen oder Verzögerungen. Gleichzeitig zeigt sich jedoch, dass die erreichte Nachrichtenrate bei gleicher Testkonfiguration im Vergleich zum bisher eingesetzten TIBCO-System deutlich geringer ist – etwa auf die Hälfte reduziert. Dieser Unterschied ist jedoch nicht zwingend inhärent der Middleware selbst zuzuschreiben, sondern lässt sich durch gezielte Optimierung der Speicherzugriffsstruktur sowie der Netzwerkkonfiguration potenziell deutlich verbessern.

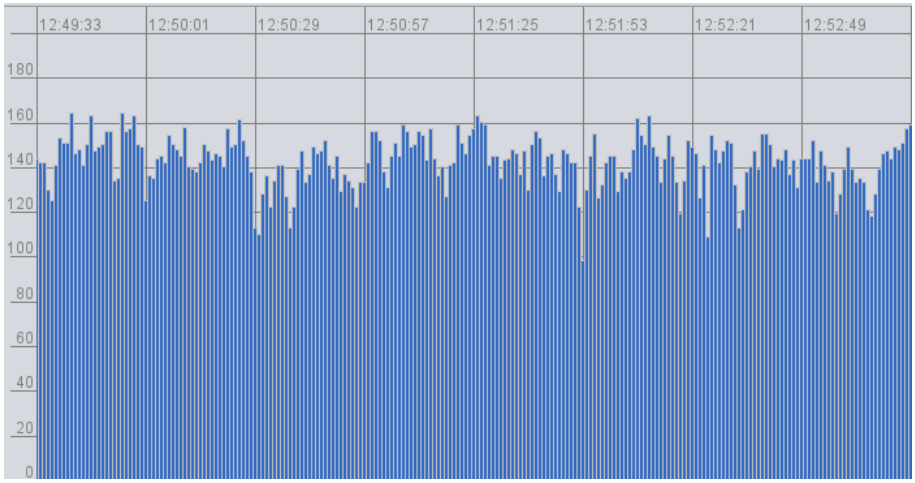



Abbildung 6.6: Nachrichten/s Dauerlast bei Artemis

### 6.2.2 Test 2: Maximallast

Auch im Szenario maximaler Systembelastung zeigt sich ein deutlich unterschiedliches Verhalten im Vergleich zum bisherigen TIBCO-System. Aufgrund der in Artemis aktivierten Persistenzmechanismen liegt die durchschnittlich erreichbare Nachrichtenrate bei lediglich 507 Nachrichten pro Sekunde und damit rund 150 Nachrichten unter dem Vergleichswert von TIBCO.

Wie bereits im Dauerlastszenario führt die persistente Zwischenspeicherung zu einem unruhigeren Verlauf der übertragenen Nachrichtenraten. In der Zeitreihe lassen sich wiederkehrende Schwankungen und vereinzelte Lastspitzen erkennen, die auf den internen Speicher- und Commit-Zyklus der Nachrichten zurückzuführen sind.

Trotz des geringeren Gesamtdurchsatzes konnte die im Rahmen der Anforderungsanalyse festgelegte Zielmarke von 365 Nachrichten pro Sekunde zuverlässig überschritten werden. Dies bestätigt, dass ActiveMQ Artemis – auch in der aktuellen Konfiguration – grundsätzlich in der Lage ist, die für die Produktion definierten Mindestanforderungen an Spitzenlast zu erfüllen. Es ist davon auszugehen, dass durch gezielte Systemoptimierungen, beispielsweise im Bereich des Message Store Backends oder der Netzwerkarchitektur, noch signifikante Leistungsreserven aktiviert werden können.

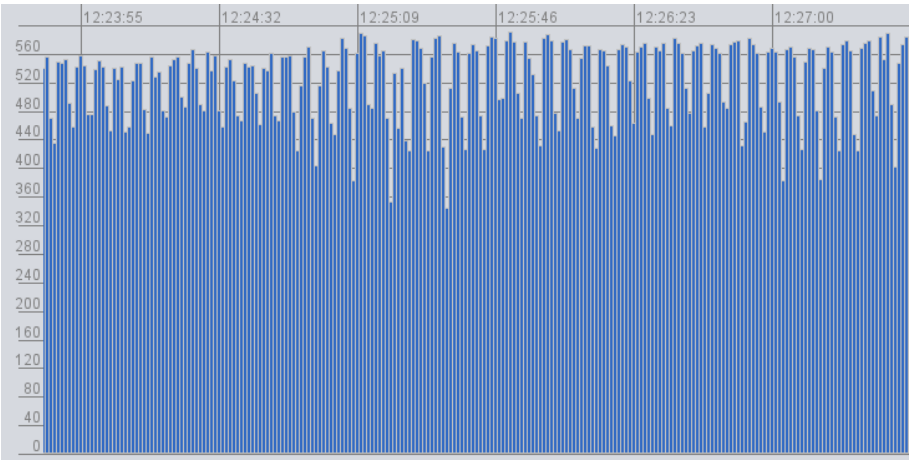



Abbildung 6.7: Nachrichten/s Hohelast bei Artemis

# 7 Ergebnisse und Diskussion

Im Rahmen dieser Arbeit wurden verschiedene MOM-Lösungen hinsichtlich ihrer Eignung als zukünftige Kommunikationsplattform für die hochautomatisierte Halbleiterfertigung unter Berücksichtigung der Anforderungen bei X-FAB evaluiert. Ziel war es, ein System zu identifizieren, welches langfristig das bestehende TIBCO-basierte Nachrichtensystem ersetzen kann und dabei sowohl funktionale als auch nicht-funktionale Anforderungen erfüllt. Die Untersuchung erfolgte anhand zuvor definierter Bewertungskriterien, darunter Leistungsfähigkeit, Zuverlässigkeit, Standardunterstützung, Skalierbarkeit, Erweiterbarkeit sowie Wartungs- und Betriebskosten.

## 7.1 Zusammenfassung der Ergebnisse

Die Analyse und die durchgeführten Leistungstests haben gezeigt, dass **ActiveMQ Artemis** in den zentralen Evaluationsdimensionen überzeugen konnte. Die wichtigsten Erkenntnisse lassen sich wie folgt zusammenfassen:

- **Zuverlässigkeit und Ausfallsicherheit:** Durch die standardmäßig aktivierte Persistenz gewährleistet Artemis eine verlustfreie Nachrichtenübertragung – auch unter hoher Systemlast. Ausfälle einzelner Komponenten führen nicht zum Verlust von Nachrichten, sondern zu temporären Verzögerungen, was für eine Produktionsumgebung mit hohen Anforderungen an Datensicherheit essenziell ist.
- **Leistung und Stabilität:** Zwar wurde in den Testreihen ein gegenüber TIBCO geringerer Durchsatz beobachtet, dieser lag jedoch deutlich über den im Realbetrieb auftretenden Peak-Anforderungen. Die gemittelte maximale Nachrichtenrate von

ca. 500 Nachrichten pro Sekunde überschreitet die geforderte Mindestlast von 365 Nachrichten pro Sekunde signifikant. Nach weiterer Leistungsoptimierung sind auch die 4-Fache Leistung erreichbar, was Zukunftssicherheit bei immer weiter steigender Transaktionslast und Produktion schafft.

- **Skalierbarkeit und Modularität:** Artemis erlaubt eine flexible Clusterbildung, unterstützt Load-Balancing und Failover-Mechanismen und bietet darüber hinaus eine moderne, erweiterbare Architektur. Damit ist es für zukünftige Wachstumsszenarien und Systemerweiterungen geeignet.
- **Standardkonformität:** Die Unterstützung von JMS 2.0 und die Integration moderner Kommunikationsprotokolle (z.B. AMQP, MQTT5) ermöglichen eine zukunftssichere Anbindung bestehender und zukünftiger Systeme.
- **Wartbarkeit und Betrieb:** Durch die Open-Source-Natur, die aktive Community und die umfangreiche Dokumentation ist Artemis im Vergleich zu proprietären Lösungen wie TIBCO wartungsfreundlicher und kosteneffizienter im Betrieb.

## 7.2 Begründung der Auswahl

Die Wahl von ActiveMQ Artemis als Ersatzlösung basiert auf der fundierten technischen Evaluierung sowie der Berücksichtigung betrieblicher Rahmenbedingungen bei X-FAB. Artemis bietet ein ausgewogenes Verhältnis zwischen Performance, Sicherheit, Flexibilität und Zukunftsfähigkeit. Auch wenn die maximale Nachrichtenrate im Vergleich zu TIBCO geringer ausfällt, stellt dieser Aspekt aufgrund der gegebenen Reserven keinen Ausschlussfaktor dar.

Tabelle 5: Bewertung der MOM-Systeme anhand des Anforderungskatalogs

| **Anforderung** | **Artemis** | **NATS** | **Kafka** | **Gewicht** |
|---|---|---|---|---|
| Unterstützung von JMS | 5 (25) | 0 (0) | 0 (0) | 5 |
| Unterstützung von MQTT | 5 (20) | 2 (6) | 2 (8) | 4 |
| Publish/Subscribe + Queue | 5 (20) | 5 (20) | 4 (16) | 4 |
| Latenz <10 ms | 4 (20) | 5 (25) | 3 (15) | 5 |
| >400 Nachrichten/s | 5 (25) | 5 (25) | 5 (25) | 5 |
| Failover ohne Datenverlust | 2 (10) | 4 (20) | 5 (25) | 5 |
| Automatische Wiederverbindung | 3 (12) | 5 (20) | 4 (16) | 4 |
| Clustering | 4 (16) | 5 (20) | 5 (20) | 4 |
| Einfache Konfiguration | 3 (9) | 4 (12) | 2 (6) | 3 |
| Integration Monitoring | - | - | - | 3 |
| Dokumentation/Community | 4 (8) | 3 (6) | 5 (10) | 2 |
| **Gesamtscore** | **185** | **154** | **141** | |

Tabelle 5 fasst die in Kapitel 3.5 definierten funktionalen und nicht-funktionalen Anforderungen in strukturierter Form zusammen. Neben der reinen Erfüllung der Kriterien durch die einzelnen Middleware-Lösungen wird zusätzlich die jeweilige Gewichtung berücksichtigt, um eine objektive Vergleichbarkeit zu ermöglichen. Auf diese Weise lassen sich sowohl technische Kernfunktionen (z. B. Unterstützung von JMS oder MQTT) als auch qualitative Aspekte wie Ausfallsicherheit, Konfigurierbarkeit oder Community-Support abbilden. Da eine Integration mit bestehenden Monitoring komponenten nicht erfolgt ist, wird diese bei der Bewertung nicht mit einbezogen.

Es besteht jedoch Optimierungspotenzial: Durch Anpassung der Speicherstruktur (z. B. Einsatz eines optimierten Journal Store oder Filesystem Backends), eine gezielte Netzwerkoptimierung sowie Tuning der Konfiguration (Threading, Paging, Asynchronous Send) kann der Durchsatz weiter erhöht werden.

Ein entscheidender Vorteil von Artemis liegt in der Offenheit und Erweiterbarkeit der Plattform. Im Gegensatz zu TIBCO ist keine kostenpflichtige Lizenzierung notwendig, was insbesondere für langfristige Skalierungsvorhaben von strategischer Bedeutung ist. Auch die einfache Integration in moderne DevOps- und Container-Infrastrukturen (z. B. Kubernetes) spricht klar für den Einsatz von Artemis.

## 7.3 Abgrenzung zu alternativen Lösungen

Die im Rahmen dieser Arbeit ebenfalls betrachteten Alternativen – insbesondere NATS – bieten ebenfalls moderne Architekturkonzepte und hohe Performance. Dennoch konnten nicht alle Anforderungen vollständig erfüllt werden. So fehlte NATS in der getesteten Version die native Unterstützung für MQTT5. Auch Apache Kafka unterstützt kein MQTT5 oder JMS, weshalb es trotz seiner hohen Skalierbarkeit und Fehlertoleranz ungeeignet ist. Das ebenfalls betrachtete MQTT5 ist nur ein Protokoll und keine MOM Lösung. Artemis stellt daher die derzeit ausgewogenste Lösung dar, die sowohl klassische Enterprise-Messaging-Funktionalität als auch moderne Integrationsmöglichkeiten bietet.

# 8 Zusammenfassung und Ausblick

Zusammenfassend lässt sich festhalten, dass ActiveMQ Artemis sämtliche für die Zielarchitektur erforderlichen Funktionalitäten bereitstellt und zugleich ein hohes Maß an Zukunftssicherheit gewährleistet. Basierend auf den durchgeführten Tests sowie den identifizierten strukturellen Vorteilen stellt Artemis somit eine geeignete und nachhaltige Nachfolgelösung für das bestehende, TIBCO-basierte System dar.

Für eine abschließende Bewertung der Systemstabilität ist jedoch ein Langzeittest unter realistischen Produktionsbedingungen erforderlich. Dieser ermöglicht die Identifikation potenzieller Engpässe, Speicherlecks oder Leistungsabfälle, die sich erst über längere Betriebszeiten hinweg manifestieren können. Darüber hinaus zeigt sich Optimierungsbedarf in der aktuellen Systemkonfiguration: Insbesondere der Wiederverbindungsprozess (*Reconnect*) zwischen Client und Broker ist in Failover-Szenarien noch zu zeitintensiv, was zu verlängerten Kommunikationsunterbrechungen führt. Eine Reduzierung dieser Wiederverbindungsdauer ist für den produktiven Einsatz von hoher Relevanz.

Nach erfolgreichem Bestehen des Langzeittests sowie der Umsetzung der identifizierten Optimierungen empfiehlt sich eine schrittweise Migration des Systems, begleitet von kontinuierlichen Performance-Messungen und Anpassungen. Auf diese Weise kann ein reibungsloser Übergang gewährleistet und die langfristige Betriebssicherheit sichergestellt werden.

# Literatur

[1] Alsharif, Khaled ; Lee, Min ; Park, Sung: A Novel MQTT 5.0-Based Over-the-Air Updating Architecture Facilitating Stronger Security. In: *Electronics* 11 (2022), Nr. 23, S. 3899. http://dx.doi.org/10.3390/electronics11233899. – DOI 10.3390/electronics11233899

[2] Apache Software Foundation: *Apache Avro Specification*. https://avro.apache.org/docs/, 2018. – Abgerufen am 18.06.2025

[3] Apache Software Foundation: *ActiveMQ Artemis*. https://activemq.apache.org/components/artemis/documentation/latest/, 2023. – Letzter Zugriff: Mai 2025

[4] Apache Software Foundation: *Apache Kafka Documentation*. https://kafka.apache.org/documentation/. Version: 2023. – Zugriff am 19.08.2025

[5] Apache Software Foundation: *Apache ActiveMQ Artemis Architecture*. https://activemq.apache.org/components/artemis/documentation/2.17.0/architecture.html. Version: unknown. – Abgerufen am 29.04.2025

[6] Appel, Stefan ; Doe, John ; Smith, Jane: Benchmarking Message Queues: A Comparative Study of Redis, ActiveMQ Artemis, RabbitMQ, and Apache Kafka. In: *Telecom* 4 (2023), Nr. 2, 298–312. http://dx.doi.org/10.3390/telecom4020018. – DOI 10.3390/telecom4020018

[7] Astrodust: *Apache ActiveMQ – Key Differences on Classic vs Artemis*. https://medium.com/@astrodust/apache-activemq-key-differences-on-classic-vs-artemis-a22efe9ea45a, 2022. – Abgerufen am 16.04.2025

[8] Banavar, G et a.: *A Case for Message Oriented Middleware*. IBM Research Division, 1999. – ISBN 0470862068

[9] Bray, Tim ; Hollander, Dave ; Layman, Andrew: *The JavaScript Object Notation (JSON) Data Interchange Format*. https://tools.ietf.org/html/rfc8259, 2017. – Abgerufen am 27.05.2025

[10] Curry, E: *Message-Oriented Middleware*. John Wiley & Sons, Ltd, 2004

[11] Eclipse Foundation: *Jakarta Messaging Specification, Version 2.0*. https://jakarta.ee/specifications/messaging/2.0/, 2020. – Abgerufen am 07.04.2025

[12] GitHub-Nutzer: *Support MQTTv5.* https://github.com/nats-io/nats-server/issues/3369, 2022. – Feature Request zur Unterstützung von MQTT 5 im NATS Server - Abgerufen am 29.04.2025

[13] GmbH, Systema: *Future Fab Message Bus.* 2023. – Interne Unterlage, nicht öffentlich zugänglich

[14] Goel, Sushant ; Sharda, Hema ; Taniar, David ; Bohme, Thomas (Hrsg.) ; Heyer, Gerhard (Hrsg.) ; Unger, Herwig (Hrsg.): *Message-oriented-middleware in a distributed environment.* Springer, 2003 (Lecture Notes in Computer Science). – 93–103 S. – ISBN 9783540204367. – International Workshop on Innovative Internet Community Systems (IICS) ; Conference date: 01-01-2003

[15] Google: *Protocol Buffers.* https://developers.google.com/protocol-buffers, 2024. – Abgerufen am 18.06.2025

[16] HiveMQ: *MQTT 5 Essentials – Introduction to MQTT 5.* https://www.hivemq.com/blog/mqtt5-essentials-part1-introduction-to-mqtt-5/. Version: 2023. – Letzter Zugriff: April 2025

[17] Horn, Torsten: *Java EE – Java Message Service (JMS).* https://www.torsten-horn.de/techdocs/jee-jms.htm, 2023. – Abgerufen am 07.04.2025

[18] Inc., Synadia C.: *NATS: The lightweight, high-performance messaging system for modern distributed systems.* https://synadia.com. Version: 2021. – Abgerufen am 27.04.2025

[19] Insights, HG: *NATS Usage Statistics.* https://discovery.hgdata.com/product/nats. Version: 2024. – Abgerufen am 27.04.2025

[20] Koenig, Reto E. ; Laederach, Lukas ; Allmen, Cédric von: How to Authenticate MQTT Sessions Without Channel- and Broker Security. In: *Proceedings of the 2019 IEEE Conference on Communications and Network Security*, 2019, S. 1–6

[21] Kreps, Jay ; Narkhede, Neha ; Rao, Jun: Kafka: a Distributed Messaging System for Log Processing. In: *Proceedings of the NetDB* ACM, 2011

[22] NATS: *NATS Docker Image Size and Deployment.* https://docs.nats.io/running-a-nats-service/nats_docker. Version: 2023. – Abgerufen am 27.04.2025

[23] NATS: *NATS High Throughput Capabilities.* https://nats.io/performance. Version: 2023. – Abgerufen am 27.04.2025

[24] NATS: *Scaling NATS: Clusters, Superclusters, and Self-healing Mechanisms*. https://docs.nats.io/running-a-nats-service/configuration/clustering. Version: 2023. – Abgerufen am 29.04.2025

[25] NATS: *Security and Access Control in NATS*. https://docs.nats.io/nats-concepts/security. Version: 2023. – Abgerufen am 01.05.2025

[26] NATS: *Supported Programming Languages and Clients*. https://docs.nats.io/nats-concepts/overview/compare-nats#language-and-platform-coverage. Version: 2023. – Abgerufen am 29.04.2025

[27] OASIS: *MQTT Version 3.1.1*. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html. Version: 2014. – Abgerufen am 07.04.2025

[28] OASIS MQTT Technical Committee: *MQTT Version 5.0 – OASIS Standard*. https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html, 2019. – Letzter Zugriff: April 2025

[29] Organization, OASIS S.: *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0*. https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-overview-v1.0.html. Version: 2012. – Abgerufen am 29.07.2025

[30] Pivotal Software, Inc.: *RabbitMQ Documentation*. https://www.rabbitmq.com, 2020. – Letzter Zugriff: April 2025

[31] Project, STOMP: *STOMP Protocol Specification, Version 1.2*. https://stomp.github.io/stomp-specification-1.2.html. Version: 2012. – Abgerufen am 29.07.2025

[32] Smith, John ; Patel, Anika: On MQTT Scalability in the Internet of Things: Issues, Solutions, and Future Directions. In: *International Journal of Distributed Sensor Networks* 18 (2022), Nr. 4, S. 1–15. http://dx.doi.org/10.1177/15501477221092345. – DOI 10.1177/15501477221092345

[33] W3C XML Schema Working Group: *XML Schema Part 1: Structures (Second Edition)*. https://www.w3.org/TR/xmlschema-1/, 2004. – Abgerufen am 27.05.2025

[34] W3C XML Working Group: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. https://www.w3.org/TR/xml/, 2008. – Abgerufen am 18.05.2025

[35] White, Richard: *Java Message Service*. O'Reilly Media, 2011

[36] XFab GmbH: *X-Fab Dresden auf einen Blick*. https://www.xfabulous.com/de/unsere-standorte/dresden/, unknown. – Abgerufen am 07.04.2025

[37] YAML Working Group: *YAML Ain't Markup Language (YAML™) Version 1.2*. `https://yaml.org/spec/1.2/spec.html`, 2009. – Abgerufen am 18.06.2025

# A Anhang

```java
/**
 * ListenerSaver is a service that asynchronously stores incoming bus messages
 * into an XML file. Messages are queued and written to the file by a background thread,
 * ensuring non-blocking event handling.
 */
public class ListenerSaver extends ASysListenService {

  private static final long serialVersionUID = -5493954474309270465L;
  private static final ExecutorService EXECUTOR = Executors.newFixedThreadPool(1);
  private static final BlockingQueue<String> MESSAGE_QUEUE = new LinkedBlockingQueue<>();

  private String FILE_PATH;

  /**
   * Initializes the service, loads the file path from configuration,
   * creates the XML file if it does not exist, and starts the message processor thread.
   *
   * @param server the system server instance
   * @param config the service configuration item
   * @throws ASysException if the service fails to initialize
   */
  @Override
  public void init(ISysServer server, ISysConfigItem config) throws ASysException {
    super.init(server, config);
    FILE_PATH = getServer().getMergedConfiguration().getValueD(String.class, "C:\\temp\\bus-messages.xml",
        "BusTest", "savePath");
    initializeFile();
    startQueueProcessor();
  }

  /**
   * Handles an incoming message. The message is timestamped and formatted,
   * then added to the message queue for asynchronous file storage.
   *
   * @param message the incoming system message
   * @param subject the subject/topic of the message
   * @throws ASysException if message handling fails
   */
  @Override
  protected void handleEventNow(ISysMessage message, ISysSubject subject) throws ASysException {
    String timestamp = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").format(new Date());
    saveMessageToQueue(message.toString(), timestamp, subject);
  }

  /**
   * Creates the XML file and writes an empty root element if the file does not exist.
   */
  private void initializeFile() {
```

```
49      File file = new File(FILE_PATH);
50
51      if (!file.exists()) {
52        try (BufferedWriter writer = new BufferedWriter(new FileWriter(
              file))) {
53          writer.write("<root>\n</root>\n");
54        } catch (IOException e) {
55          e.printStackTrace();
56        }
57      }
58    }
59
60    /**
61     * Formats the message and adds it to the message queue.
62     *
63     * @param message   the raw message string
64     * @param timestamp the timestamp of when the message was received
65     * @param subject   the message's subject
66     */
67    private void saveMessageToQueue(String message, String timestamp,
          ISysSubject subject) {
68      String formattedMessage = formatMessage(message, timestamp,
            subject);
69      MESSAGE_QUEUE.add(formattedMessage);
70    }
71
72    /**
73     * Enhances the raw XML message by adding a timestamp and subject
          information.
74     *
75     * @param message   the original message string (expected to start
          with &lt;message&gt;)
76     * @param timestamp the formatted timestamp to include
77     * @param subject   the subject/topic associated with the message
78     * @return the formatted XML message as a string
79     * @throws IllegalArgumentException if the message format is
          invalid
80     */
81    private String formatMessage(String message, String timestamp,
          ISysSubject subject) {
82      if (!message.startsWith("<message")) {
83        throw new IllegalArgumentException("Invalid XML format: Message
              must start with <message>");
84      }
85
86      int insertPosition = message.indexOf(">");
87      String messageWithTimestamp = message.substring(0, insertPosition
          ) + " timestamp=\"" + timestamp + "\""
88          + message.substring(insertPosition);
89
90      return messageWithTimestamp.replaceFirst("(?<=<message[^>]*>)", "
          \n<subject>" + subject + "</subject>\n");
91    }
92
93    /**
94     * Starts a background thread that continuously polls the queue and
          writes messages to the file.
95     * This keeps file operations asynchronous and prevents blocking
          the main message handler.
96     */
97    private void startQueueProcessor() {
```

```
98      EXECUTOR.submit(() -> {
99        while (!Thread.currentThread().isInterrupted()) {
100         try {
101           String message = MESSAGE_QUEUE.poll(1, TimeUnit.SECONDS);
102           if (message != null) {
103             appendMessageToFile(message);
104           }
105         } catch (Exception e) {
106           e.printStackTrace(); // Ideally use logging
107         }
108       }
109     });
110   }
111
112   private final Object FILE_LOCK = new Object();
113
114   /**
115    * Appends the given message to the XML file, inserting it just
           before the closing root tag.
116    * This operation is synchronized to prevent concurrent file access
           .
117    *
118    * @param message the message to write to the file
119    * @throws IOException if an I/O error occurs during file writing
120    */
121   private void appendMessageToFile(String message) throws IOException
           {
122     synchronized (FILE_LOCK) {
123       File file = new File(FILE_PATH);
124       String closingTag = "</root>";
125       byte[] closingBytes = ("</root>" + System.lineSeparator()).
              getBytes();
126
127       try (RandomAccessFile raf = new RandomAccessFile(file, "rw")) {
128         long fileLength = raf.length();
129         long pos = fileLength - closingBytes.length;
130
131         raf.seek(pos); // Position cursor before </root>
132         raf.writeBytes(message + System.lineSeparator());
133         raf.writeBytes(closingTag + System.lineSeparator());
134       }
135     }
136   }
137 }
```

Listing 4: ListenerSaver – Java-Klasse zum Speichern von Nachrichten

```java
/**
 * The EventSender class reads pre-recorded XML messages from a file and
 * replays them asynchronously into the message bus, preserving their original
 * timing structure. This is useful for simulation, testing, or replaying real-world message traces.
 */
public class EventSender extends ASysListenService {

  private static final long serialVersionUID = 3550176584946238534L;
  private static final ScheduledExecutorService EXECUTOR = Executors.newScheduledThreadPool(1);
  private String FILE_PATH;
  private long timemanipulation = 1;
  transient ISysSubject topic;
  int count = 0;

  /**
   * Initializes the service by reading the configured message file path.
   *
   * @param server the server instance
   * @param config the configuration item containing service parameters
   * @throws ASysException if initialization fails
   */
  @Override
  public void init(ISysServer server, ISysConfigItem config) throws ASysException {
    super.init(server, config);
    FILE_PATH = getServer().getMergedConfiguration().getValueD(String.class, "C:\\temp\\bus-messages.xml",
        "BusTest", "readPath");
  }

  /**
   * Called upon an event. Triggers the asynchronous replay of messages.
   *
   * @param message the received system message
   * @param subject the associated subject/topic
   * @throws ASysException if message handling fails
   */
  @Override
  protected void handleEventNow(ISysMessage message, ISysSubject subject) throws ASysException {
    replayMessages();
  }

  /**
   * Starts asynchronous replay of all messages read from the XML file.
   */
  private void replayMessages() {
    CompletableFuture.runAsync(() -> {
      try {
        List<MessageData> messages = readMessagesFromFile();
        simulateMessageReplay(messages);
      } catch (Exception e) {
```

```
50          e.printStackTrace(); // Consider using a logger
51        }
52      }, EXECUTOR);
53    }
54
55    /**
56     * Reads and parses message entries from the XML file into
          structured data.
57     *
58     * @return a list of parsed messages
59     * @throws IOException if reading the file fails
60     */
61    private List<MessageData> readMessagesFromFile() throws IOException
          {
62      List<MessageData> messages = new ArrayList<>();
63      Pattern timestampPattern = Pattern.compile("timestamp
          =\\\"([^\\\"]+)\\\"");
64      Pattern subjectPattern = Pattern.compile("<subject>(.+?)</subject
          >");
65
66      try (BufferedReader reader = new BufferedReader(new FileReader(
          FILE_PATH))) {
67        String line, topic = null, timestamp = null;
68        StringBuilder messageBuffer = new StringBuilder();
69
70        while ((line = reader.readLine()) != null) {
71          if (line.startsWith("<message")) {
72            if (messageBuffer.length() > 0) {
73              messages.add(new MessageData(messageBuffer.toString(),
                    timestamp, topic));
74              messageBuffer.setLength(0);
75            }
76            timestamp = extractValue(line, timestampPattern);
77            topic = null;
78          }
79          if (line.contains("<subject>")) {
80            topic = extractValue(line, subjectPattern);
81          }
82          if (!line.endsWith("root>")) {
83            messageBuffer.append(line).append(System.lineSeparator());
84          }
85        }
86        if (messageBuffer.length() > 0) {
87          messages.add(new MessageData(messageBuffer.toString(),
                timestamp, topic));
88        }
89      }
90      return messages;
91    }
92
93    /**
94     * Extracts a regex-matched value from a line.
95     *
96     * @param line    the input line to search
97     * @param pattern the regex pattern to apply
98     * @return the extracted value, or null if not found
99     */
100   private String extractValue(String line, Pattern pattern) {
101     Matcher matcher = pattern.matcher(line);
102     return matcher.find() ? matcher.group(1) : null;
103   }
```

```
104
105   /**
106    * Replays the messages in their original time order, with optional
           time manipulation factor.
107    *
108    * @param messages list of messages to replay
109    */
110   private void simulateMessageReplay(List<MessageData> messages) {
111     messages.sort(Comparator.comparingLong(MessageData::
          getTimestampAsMillis));
112     long baseTimestamp = messages.get(0).getTimestampAsMillis();
113
114     for (MessageData message : messages) {
115       long delay = (message.getTimestampAsMillis() - baseTimestamp) *
            timemanipulation;
116       EXECUTOR.schedule(() -> sendMessage(message), delay, TimeUnit.
            MILLISECONDS);
117     }
118   }
119
120   private String meskey = "abc";
121   private String curmeskey = "abc";
122
123   /**
124    * Sends a single message to the bus system, if it has not already
           been sent with the same key.
125    *
126    * @param messageData the message data to send
127    */
128   private void sendMessage(MessageData messageData) {
129     try {
130       ISysMessageItem message = FSysMessage.createMessage();
131       message.addItem("Message", messageData.getMessage());
132       message.addItem("time", messageData.getTimestamp());
133       message.addItem("counter", count);
134
135       curmeskey = messageData.getTimestamp();
136       topic = getServer().getSubjectManager().createSubjectFactory("
            standard").createSubject(null);
137       topic.setLevels(messageData.getTopic());
138       topic.addLevel(0, "Test");
139
140       if (!curmeskey.contains(meskey)) {
141         getServer().getBusManager().get().get(0).publish(topic, new
              CSysBusSendMessage(message));
142         meskey = curmeskey;
143       }
144     } catch (Exception e) {
145       e.printStackTrace(); // Consider using proper logging
146     }
147   }
148
149   /**
150    * Inner data structure to represent a parsed message with its
           metadata.
151    */
152   private static class MessageData {
153     private final String message;
154     private final String timestamp;
155     private final String topic;
156
```

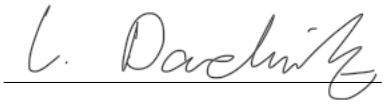
```
157     /**
158      * Constructs a new MessageData instance.
159      *
160      * @param message   the raw XML content
161      * @param timestamp the timestamp extracted from the message
162      * @param topic     the message subject/topic
163      */
164     public MessageData(String message, String timestamp, String topic
           ) {
165       this.message = message;
166       this.timestamp = timestamp;
167       this.topic = topic;
168     }
169
170     /**
171      * @return the raw message content
172      */
173     public String getMessage() {
174       return message;
175     }
176
177     /**
178      * @return the timestamp string
179      */
180     public String getTimestamp() {
181       return timestamp;
182     }
183
184     /**
185      * @return the subject/topic
186      */
187     public String getTopic() {
188       return topic;
189     }
190
191     /**
192      * Converts the timestamp string into milliseconds since epoch.
193      *
194      * @return the timestamp in milliseconds
195      */
196     public long getTimestampAsMillis() {
197       try {
198         return new SimpleDateFormat("yyyy-MM-dd HH:mm:ss.SSS").parse(
              timestamp).getTime();
199       } catch (ParseException e) {
200         return 0;
201       }
202     }
203   }
204 }
```

Listing 5: EventSender – Java-Klasse zum Senden gespeicherter Nachrichten

# Erklärung

Der Verfasser erklärt, dass er die vorliegende Arbeit selbständig, ohne fremde Hilfe und ohne Benutzung anderer als die angegebenen Hilfsmittel angefertigt hat. Die aus fremden Quellen (einschließlich elektronischer Quellen) direkt oder indirekt übernommenen Gedanken sind ausnahmslos als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Zudem bestätigt der Verfasser, dass er den Lehrstuhlleitfaden in der jeweiligen geltenden Fassung gelesen und verstanden hat und sich daher über die gestellten Anforderungen und Bewertungsmaßstäbe im Klaren ist.

28.08.2025, Dresden



Datum, Ort

Lucas Dachwitz