HIGH-PERFORMANCE NEAR-TIME PROCESSING OF BULK DATA

MARTIN SWIENTEK

RESEARCH WITH PLYMOUTH UNIVERSITY

A thesis submitted the Plymouth University
in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

July 2014 – version 0.1

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

SLA Service Level Agreements

Part I

FIELD OF RESEARCH

1 INTRODUCTION

Enterprise Systems like customer-billing systems or financial transaction systems are required to process large volumes of data in a fixed period of time. For example, a billing system for a large telecommunication provider has to process more than 1 million bills per day. Those systems are increasingly required to also provide near-time processing of data to support new service offerings.

Traditionally, enterprise systems for bulk data processing are implemented as batch processing systemsFleck (1999). Batch processing delivers high throughput but cannot provide near-time processing of data, that is the end-to-end latency of such a system is high. End-to-end latency refers to the period of time that it takes for a business process, implemented by multiple subsystems, to process a single business event. For example, consider the following billing system of telecommunications provider:

- Customers are billed once per month
- Customers are partitioned in 30 billing groups
- The billing system processes 1 billing group per day, running 24h under full load.

In this case, the mean time for a call event to be billed by the billing system is 1/2 month. That is, the mean end-to-end latency of this system is 1/2 month.

A lower end-to-end latency can be achieved by using single-event processing, for example by utilizing a message-oriented middleware for the integration of the services that form the enterprise system. While this approach is able to deliver near-time processing, it is hardly capable for bulk data processing due to the additional communication overhead for each processed message. Therefore, message-based processing is usually not considered for building a system for bulk data processing requiring high throughput.

The processing type is usually a fixed property of an enterprise system that is decided when the architecture of the system is designed, prior to implementing the system. This choice depends on the non-functional requirements of the system. These requirements are not fixed and can change during the lifespan of a system, either anticipated or not anticipated.

Additionally, enterprise systems often need to handle load peaks that occur infrequently. For example, think of a billing system with moderate load over most of the time, but there are certain events

with very high load such as New Year's Eve. Most of the time, a low end-to-end latency of the system is preferable when the system faces moderate load. During the peak load, it is more important that the system can handle the load at all. A low end-to-end latency is not as important as an optimized maximum throughput in this situation.

1.1 RESEARCH PROBLEM

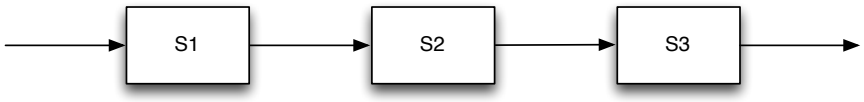
1.2 AIMS AND OBJECTIVES OF THE RESEARCH

1.3 CONTRIBUTIONS

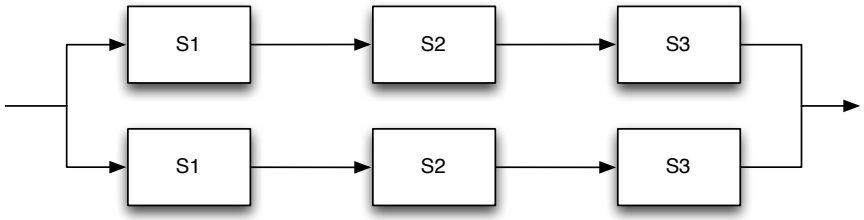
1.4 OUTLINE OF THE THESIS

2 BACKGROUND

We consider a distributed system for bulk data processing consisting of several subsystems running on different nodes that together form a processing chain, that is, the output of subsystem S1 is the input of the next subsystem S2 and so on (see Figure 27a).



(a) Single processing line



(b) Parallel processing lines

Figure 1: A system consisting of several subsystems forming a processing chain

To facilitate parallel processing, the system can consist of several lines of subsystems with data beeing distributed among each line. For simplification, we consider a system with a single processing line in the remainder of this paper.

We discuss two processing types for this kind of system, batch processing and message-based processing.

2.1 BATCH PROCESSING

The traditional operation paradigm of a system for bulk data processing is batch processing (see Figure 28). A batch processing system is an application that processes bulk data without user interaction. Input and output data is usually organised in records using a file- or database-based interface. In the case of a file-based interface, the application reads a record from the input file, processes it and writes the record to the output file.

A batch processing system exhibits the following key characteristics:

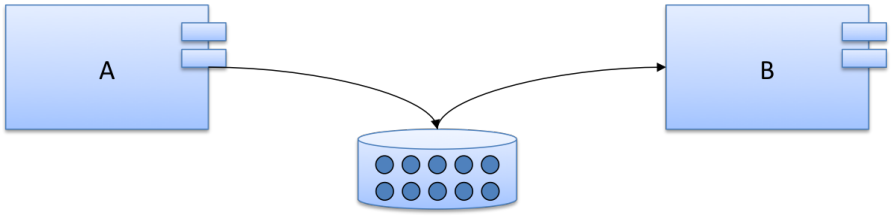


Figure 2: Batch processing

- **Bulk processing of data**
  A Batch processing system processes several gigabytes of data in a single run thus providing a high throughput. Multiple systems are running in parallel controlled by a job scheduler to speed up processing. The data is usually partitioned and sorted by certain criteria for optimized processing. For example, if a batch only contains data for a specific product, the system can pre-load all necessary reference data from the database to speed up the processing.

- **No user interaction**
  There is no user interaction needed for the processing of data. It is impossible due to the amount of data being processed.

- **File- or database-based interfaces**
  Input data is read from the file system or a database. Output data is also written to files on the file system or a database. Files are transferred to the consuming systems through FTP by specific jobs.

- **Operation within a limited timeframe**
  A batch processing system often has to deliver its results in a limited timeframe due to Service Level Agreements (SLA) with consuming systems.

- **Offline handling of errors**
  Erroneous records are stored to a specific persistent memory (file or database) during operation and are processed afterwards.

Applications that are usually implemented as batch processing systems are billing systems for telecommunication companies used for mediating, rating and billing of call events.

2.2 MESSAGE-BASE PROCESSING

Messaging facilitates the integration of heterogeneous applications using asynchronous communication. Applications are communicating with each other by sending messages (see Figure 29). A messaging server or message-oriented middleware handles the asynchronous ex-

change of messages including an appropriate transaction control Conrad et al. (2006).

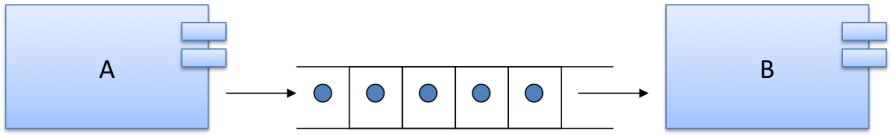


Figure 3: Message-based processing

Hohpe et al.Hohpe and Woolf (2003) describe the following basic messaging concepts:

- **Channels**
  Messages are transmitted through a channel. A channel connects a message sender to a message receiver.

- **Messages**
  A message is packet of data that is transmitted through a channel. The message sender breaks the data into messages and sends them on a channel. The message receiver in turn reads the messages from the channel and extracts the data from them.

- **Pipes and Filters**
  A message may pass through several processing steps before it reaches its final destination. Multiple processing steps are chained together using a pipes and filters architecture.

- **Routing**
  A message may have to go through multiple channels before it reaches its destination. A message router acts as a filter and is capable of routing a message to the next channel or to another message router.

- **Transformation**
  A message can be transformed by a message translator if the message sender and receiver do not agree on the format for the same conceptual data.

- **Endpoints**
  A message endpoint is a software layer that connects arbitrary applications to the messaging system.

Message-based systems are able to provide near-time processing of data due to their lower latency compared with batch processing systems. The advantage of a lower latency comes with a performance cost in regard to a lower throughput because of the additional overhead for each processed message. Every message needs amongst others to be serialised and deserialised, mapped between different protocols and routed to the appropriate receiving system.

2.3 LATENCY VS. THROUGHPUT

Throughput and latency are performance metrics of a system. The following definitions of throughput and latency are used in this paper:

- **Maximum Throughput**
  The number of events the system is able to process in a fixed timeframe.
- **Ent-to-end Latency**
  The period of time between the occurrence of an event and its processing. End-to-end latency refers to the total latency of a complete business process implemented by multiple subsystems. The remainder of this paper focusses on end-to-end latency using the general term latency as an abbreviation.

2.3.1 *Batch processing*

A business process, such as billing, implemented by a system using batch processing exhibits a high end-to-end latency. For example, consider the following billing system:

- Customers are billed once per month
- Customers are partitioned in 30 billing groups
- The billing system processes 1 billing group per day, running 24h under full load.

In this case, the mean time for a call event to be billed by the billing system is 1/2 month. That is, the mean end-to-end latency of this system is 1/2 month.

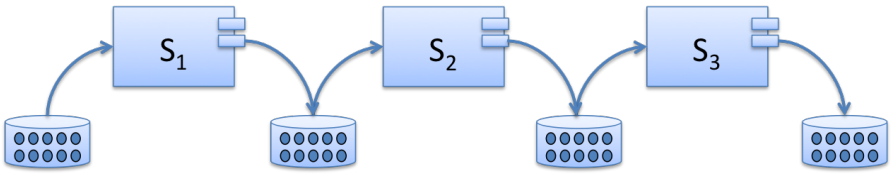


Figure 4: Batch processing system comprised of three subsystems

Assuming the system S_{Batch} which is comprised of N subsystems $S_1, S_2, \ldots, S_N$ (see Figure 4 for an example with $N = 3$):

$$S_{Batch} = \{S_1, S_2, \ldots, S_N\}$$

The subsystem S_i reads its input data from the database DB_i in one chunk, processes it and writes the output to the database DB_{i+1}. When S_i has finished the processing, the next subsystem S_{i+1} reads the input data from DB_{i+1}, processes it and writes the output to

DB_{i+2}, which in turn is read and processed from subsystem S_{i+3} and so on.

The latency $L_{E_{S_{Batch}}}$ of a single event processed by the system S_{Batch} is determined by the total processing time $PT_{S_{Batch}}$, which is the sum of the processing time PT_i of each subsystem S_i:

$$L_{E_{S_{Batch}}} = PT_{S_{Batch}} = \sum_{i=1}^{N} PT_i$$

where N is the number of subsystems.

The processing time PT_i of the subsystem S_i is the sum of the processing time of each event PT_{E_j} and the additional processing overhead OH_i, which includes the time spent for reading and writing the data, opening and closing transactions, etc:

$$PT_i = \left(\sum_{j=1}^{M} PT_{E_j} \right) + OH_i$$

where M is the number of events.

To allow for near-time processing, it is necessary to decrease the latency L_{E_S} of a single event. This is can be achieved by using message-based processing instead of batch processing.

2.3.2 *Message-based processing*

The subsystem S_i of a message-based system $S_{Message}$ reads a single event from its input message queue MQ_i, processes it and writes it to the output message queue MQ_{i+1}. As soon as the event is written to the message queue MQ_{i+1}, it is read by the subsystem S_{i+1}, which processes the event and writes to the message queue $MQi + 2$ and so on (see Figure 5).

The latency $L_{E_{S_{Message}}}$ of a single event processed by the system $S_{Message}$ is determined by the total processing time $PT_{E_{S_{Message}}}$ of this event, which is the sum of the processing time PT_{E_i} and the processing overhead OH_{E_i} for the event of each subsystem:

$$L_{E_{S_{Message}}} = PT_{E_{S_{Message}}} = \sum_{i+1}^{N} (PT_{E_i} + OH_{E_i})$$

where N is the number of subsystems. Please note that the wait time of the event is assumed to be 0 for simplification.

The processing overhead OH_{E_i} includes amongst others the time spent for unmarshalling and marshalling, protocol mapping and opening and closing transactions, which is done for every processed event.

Since the processing time $PT_{E_{S_{Message}}}$ of a single event is much shorter than the total processing time $PT_{S_{Batch}}$ of all events, the latency $L_{E_{S_{Message}}}$ of a single event using a message-based system is

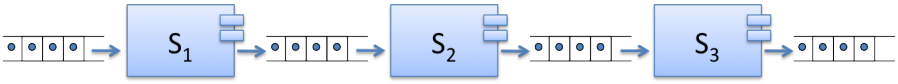


Figure 5: Message-based system comprised of three subsystems

much smaller than the latency $L_{E_{S_{Batch}}}$ of a single event processed by a batch-processing system.

$$PT_{E_{S_{Message}}} < PT_{S_{Batch}} \Rightarrow L_{E_{S_{Message}}} < L_{E_{S_{Batch}}}$$

Message-based processing adds an overhead to each processed event in contrast to batch processing, which adds a single overhead to each processing cycle. Hence, the accumulated total processing overhead $OH_{S_{Message}}$ of a message-based system $S_{Message}$ for processing m events is larger than the total processing overhead of a batch processing system:

$$OH_{S_{Message}} = \sum_{i=1}^{n} OH_{E_i} * m > OH_{S_{Batch}} = \sum_{i=1}^{n} OH_i$$

A message-based system, while having a lower end-to-end latency, is not able to process the same amount of events in the same time as a batch processing system and therefore cannot provide the same maximum throughput.

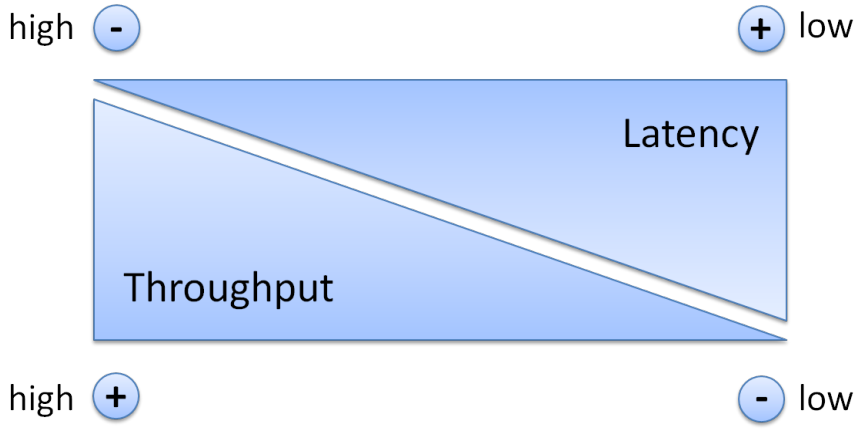


Figure 6: Latency and throughput are opposed to each other

From this follows that latency and throughput are opposed to each other (see Figure 6). High throughput, as provided by batch processing, leads to high latency, which impedes near-time processing. On the other hand, low latency, as provided by a message-based system, cannot provide the throughput needed for bulk data processing because of the additional overhead for each processed event.

2.4 SERVICE-ORIENTED ARCHITECTURE

Service-Oriented Architecture (SOA) is an architectural pattern to build application landscapes from single business components. These business components are loosely coupled by providing their functionality in form of services. A service represents an abstract business view of the functionality and hides all implementation details of the component providing the service. The definition of a service acts as a contract between the service provider and the service consumer. Services are called using a unified mechanism, which provides a platform independent connection of the business components while hiding all the technical details of the communication. The calling mechanism also includes the discovery of the appropriate service (Richter et al., 2005).

By separating the technical from the business aspects, SOA aims for a higher level of flexibility of enterprise applications.

2.5 ENTERPRISE SERVICE BUS

An Enterprise Service Bus (ESB) is an integration platform that combines messaging, web services, data transformation and intelligent routing (Schulte, 2002). Table 1 shows the main characteristics of an ESB (Chappell, 2004). All application components and integration services that are connected to the ESB are viewed as abstract service endpoints. Abstract endpoints are logical abstractions of services that are plugged into the ESB and are all equal participants (Chappell, 2004). An abstract endpoint can represent a whole application package such as a CRM or ERP system, a small web service or an integration service of the ESB such as a monitoring, logging or transformation service. As integration platform the ESB supports various types of connections for the service endpoints. These can be SOAP, HTTP, FTP, JMS or other programming APIs for C, C++, C#, etc. It is often stated that "if you can't bring the application to the bus, bring the bus to the application" (Chappell, 2004).

The backbone of the ESB is a message-oriented middleware (MOM), which provides an asynchronous, reliable and efficient transport of data between the service endpoints. The concrete protocol of the MOM, such as JMS, WS-Rel* or a proprietary protocol is thereby abstracted by the service endpoint. The ESB is thus a logical layer over the messaging middleware. The utilised protocol can also be varied by the ESB depending on the Quality of Service (QoS) requirements or deployment situations. Service endpoints can be orchestrated to process flows, which are mapped to concrete service invocations by the ESB.

The physical representation of a service endpoint is the service container. The service container is a remote process, which hosts the busi-

Pervasiveness	An ESB supports multiple protocols and client technologies. It can span an entire organisation including its business partners.
Highly distributed	An ESB integrates loosely coupled application components that form a highly distributed network.
Selective deployment of integration components	The services of an ESB are independent of each other and can be separately deployed.
Security and reliability	An ESB provides reliable messaging, transactional integrity and secure authentication.
Orchestration and process flow	An ESB supports the orchestration of application components controlled by message metadata or an orchestration language like WS-BPEL.
Autonomous yet federated managed environment	Different departments can still separately manage an ESB that spans the whole organisation.
Incremental adoption	The adoption of an ESB can be incremental one project after another.
XML support	XML is the native data format of an ESB.
Real-time insight	An ESB provides real-time throughput of data by the use of its underlying message-oriented middleware and thus decreases latency.

Table 1: Main characteristics of an ESB (Chappell, 2004)

ness or technical components that are connected through the bus. The set of all service containers therefore constitute the logical ESB.

A service container provides the following interfaces (Chappell, 2004):

- **Service interface**
  The service interface provides an entry endpoint and exit endpoint to dispatch messages to and from the service.
- **Management interface**
  The management interface provides an entry endpoint for retrieving configuration data and an exit endpoint for sending logging, event tracking and performance data.

2.6 PERFORMANCE ISSUES

This section describes the performance issues of an SOA middleware that inhibit their appropriateness for systems with high performance requirements.

2.6.1 *Distributed Architecture*

A system implemented according to the principles of SOA is a distributed system. Services are hosted on different locations belonging to different departments and even organizations. Hence, the performance drawbacks of a distributed system generally also apply to SOA. This includes the marshalling of the data that needs to be sent to the service provider by the service consumer, sending the data over the network and the unmarshalling of data by the service provider.

2.6.2 *Integration of Heterogeneous Technologies*

A main goal of introducing an SOA is to integrate applications implemented with heterogeneous technologies. This is achieved by using specific middleware and intermediate protocols for the communication. These protocols are typically based on XML, like SOAP (*SOAP Specification*, 2007). XML, as a very verbose language, adds a lot of meta-data to the actual payload of a message. The resulting request is about 10 to 20 times larger than the equivalent binary representation (O'Brien et al., 2007), which leads to a significant higher transmission time of the message. Processing these messages is also time-consuming, as they need to get parsed by a XML parser before the actual processing can occur.

The usage of a middleware like an Enterprise Service Bus (ESB) adds further performance costs. An ESB usually processes the messages during transferring. Among other things, this includes the mapping between different protocols used by service providers and ser-

vice consumers, checking the correctness of the request format, adding message-level security and routing the request to the appropriate service provider (See, for example, Josuttis (2007) or Krafzig et al. (2005)).

2.6.3 *Loose Coupling*

Another aspect of SOA that has an impact on performance is the utilisation of loose coupling. The aim of loose coupling is to increase the flexibility and maintainability of the application landscape by reducing the dependency of its components on each other. This denotes that service consumers shouldn't make any assumptions about the implementation of the services they use and vice versa. Services become interchangeable as long they implement the interface the client expects.

Engels et al. (2008) consider two components A and B loosely coupled when the following constraints are satisfied:

- **Knowlegde**
  Component A knows only as much as it is needed to use the operations offered by component B in a proper way. This includes the syntax and semantic of the interfaces and the structure of the transferred data.

- **Dependence on availability**
  Component A provides the implemented service even when component B is not available or the connection to component B is not available.

- **Trust**
  Component B does not rely on component A to comply with pre-conditions. Component A does not rely on component B to comply with post-conditions.

The gains in flexibility and maintainability of loose coupling are amongst others opposed by performance costs.

Service consumers and service provider are not bound to each other statically. Thus, the service consumer needs to determine the correct end point of the service provider during runtime. This can be done by looking up the correct service provider in a service repository either by the service consumer itself before making the call or by routing the message inside the ESB.

Apart from very few basic data types, Service consumers and service providers do not share the same data model. It is therefore necessary to map data between the data model used by the service consumer and the data model used by the service provider.

2.7 CURRENT APPROACHES FOR IMPROVING THE PERFORMANCE OF AN SOA MIDDLEWARE

This section describes current approaches to the performance issues introduced in the previous section.

2.7.1 *Hardware*

The obvious solution to improve the processing time of a service is the utilization of faster hardware and more bandwidth. SOA performance issues are often neglected by suggesting that faster hardware or more bandwidth will solve this problem. However, it is often not feasible to add faster or more hardware due to high cost pressure.

2.7.2 *Compression*

The usage of XML as an intermediate protocol for service calls has a negative impact on their transmission times over the network. The transmission time of service calls and responses can be decreased by compression. Simply compressing service calls and responses with gzip can do this. The World Wide Web Consortium (W3C) proposes a binary presentation of XML documents called binary XML (*EXI Working Group*, 2007) to achieve a more efficient transportation of XML over networks.

It must be pointed out that the utilisation of compression adds the additional costs of compressing and decompressing to the overall processing time of the service call.

2.7.3 *Service Granularity*

To reduce the communication overhead or the processing time of a service, the service granularity should be reconsidered.

Coarse-grained services reduce the communication overhead by achieving more with a single service call and should be the favoured service design principle (Hess et al., 2006). However, the processing time of a coarse grained service can pose a problem to a service consumer that only needs a fracture of the data provided by the service. To reduce the processing time it could be considered in this case to add a finer grained service that provides only the needed data (Josuttis, 2007).

It should be noted that merging multiple services to form a more coarse grained service or splitting a coarse grained service into multiple services to solve performance problems specific to a single service consumer reduces the reusability of the services for other service consumers (Josuttis, 2007).

2.7.4 *Degree of Loose Coupling*

The improvements in flexibility and maintainability gained by loose coupling are opposed by drawbacks on performance. Thus, it is crucial to find the appropriate degree of loose coupling.

Hess et al. (2006) introduce the concept of distance to determine an appropriate degree of coupling between components. The distance of components is comprised of the functional and technical distance. Components are functional distant if they share few functional similarities. Components are technical distant if they are of a different category. Categories classify different types of components like inventory components, process components, function components and interaction components.

Distant components trust each other in regard to the compliance of services levels to a lesser extent than near components do. The same applies to their common knowledge. Distant components share a lesser extent of knowledge of each other. Therefore, Hess et al. (2006) argue that distant components should be coupled more loosely than close components.

The degree of loose coupling between components that have been identified to be performance bottlenecks should be reconsidered to find the appropriate trade-off between flexibility and performance. It can be acceptable in that case to decrease the flexibility in favour of a better performance.

2.8 SUMMARY

Message-oriented middleware facilitates the integration of applications using asynchronous messages. An Enterprise Service Bus is such a middleware combining messaging, web services, data transformation and intelligent routing. Message-based systems are able to provide near-time processing of data due to their lower latency compared with batch processing systems. The advantage of a lower latency comes with a performance cost in regard to a lower throughput because of the additional overhead for each processed message. Every message needs amongst others to be serialised and deserialised, mapped between different protocols and routed to the appropriate receiving system.

Current approaches to improve the throughput performance of message-based systems try to reduce the transmission time by compressing messages. Another approach is to adjust the service granularity to form more coarse-grained services or to adjust the degree of loose coupling to reduce the communication overhead.

While these approaches generally improve the performance of message-based systems, they are still not able provide the same throughput as that can be achieved with a batch processing system. Additionally, the

current approaches are static and thus need to be considered at the design-time of the system. The next chapter presents an SOA middleware for high performance near-time processing of bulk data which is a novel approach to dynamically reduce the latency of a system while still providing high throughput.

3

RELATED WORK

Part II

CONTRIBUTIONS

4

PERFORMANCE EVALUATION OF BATCH AND MESSAGE-BASED SYSTEMS

4.1 A REAL WORLD EXAMPLE APPLICATION

In this section we introduce the two prototypes of a billing system that we have built to evaluate the performance of batch and message-based processing.

A billing system is a distributed system consisting of several sub components that process the different billing sub processes like mediation, rating, billing and presentment (see Figure 7).

Figure 7: Billing process

The mediation components receive usage events from delivery systems, like switches and transform them into a format the billing system is able to process. For example, transforming the event records to the internal record format of the rating and billing engine or adding internal keys that are later needed in the process. The rating engine assigns the events to the specific customer account, called guiding, and determines the price of the event, depending on the applicable tariff. It also splits events if more than one tariff is applicable or the customer qualifies for a discount. The billing engine calculates the total amount of the bill by adding the rated events, recurring and one-time charges and discounts. The output is processed by the presentment components, which format the bill, print it, or present it to the customer in self-service systems, for example on a website.

In order to compare batch and message-based types of processing, two different prototypes of a billing application have been developed. Each prototype implements the mediation and rating steps of the billing process. Figure 8 shows the components of the billing prototype:

- **Event Generator**
  The *Event Generator* generates the calling events, i.e. the call detail records (CDR) that are processed by the billing application.

- **Mediation**
  The *Mediation* component checks wether the calltime of the calldetail record exceeds the minimal billable length or if it belongs to a flatrate account and sets the corresponding flags of the record. The output of the *Mediation* component are normalized

call records (NCDR) that are further processed by the *Rating* component.

- **Rating**
  The *Rating* component processes the output from the *Mediation* component. It assigns the calldetail record to a customer account and determines the price of the call event by looking up the correspondant product and tariff in the *Master Data DB*. The output of the *Rating* component (costed events) is afterwards written to the *Costed Events DB*.

- **Master Data DB**
  The *Master Data DB* contains products, tariffs and accounts used by the *Event Generator* and the *Rating* component.

- **Costed Events DB**
  The *Costed Events DB* contains the result of the *Rating* component, i.e. the costed events.

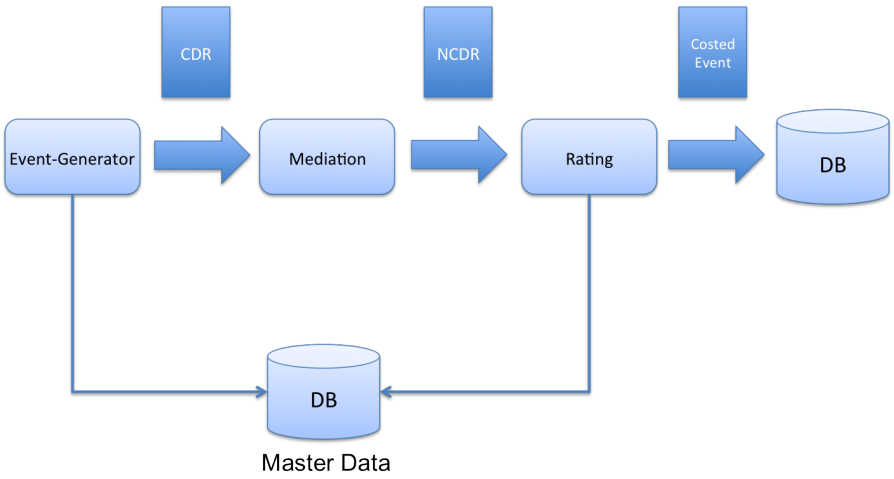


Figure 8: Components of the billing application prototype

The prototypes are implemented with Java 1.6 using JPA for the data-access layer and a MySQL database. To ensure comparability, the prototypes share the same business components, database and data-access layer, varying only in different integration layers.

4.1.1 *Batch prototype*

The batch prototype implements the billing application utilizing the batch processing type. It uses the Spring Batch framework*Spring Batch* (2013), a Java framework that facilitates the implementation of batch applications by providing basic building blocks for reading, writing and processing data.

The main entities in Spring Batch are Jobs and Steps. A Job defines the processing flow of the batch application and consists of one or

more steps. A basic step is comprised of an item reader, item processor and item writer (see Figure 9).

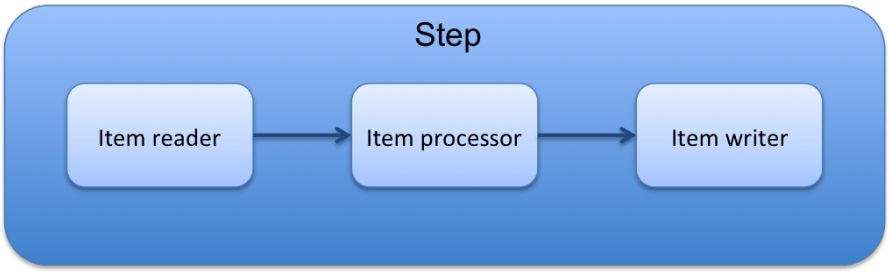


Figure 9: A Step consists of an item reader, item processor and item writer

The item reader reads records of data in chunks, for example from a file, and converts them to objects. These objects are then processed by the item processor, which contains the business logic of the batch application. Finally, the processed objects are getting written to the output destination, for example a database, by the item writer.

The mediation batch job *mediationMultiThreadedJob* consists of two steps, the *mediationMultiThreadedStep* and the *renameFileMulitThreaded-Step*. The step is multithreaded and uses 10 threads for processing. It consists of a *rawUsageMultiThreadedEventReader*, a thread safe reader implementation that reads call detail records from the input file and converts them to objects, a *rawUsageEventProcessor*, that processes the call detail objects by calling the mediation business logic and a *log-gingSimpleCdrWriter*, which writes the processd call detail objects to the output file. The step uses an commit interval of 1000, meaning that the input data is processed in chunks of 1000 records. After the input file has been processed by the *mediationMultiThreadedStep* it is getting renamed to its final name by the *renameFileMultiThreadedStep*.

Figure 10 shows the architecture of the batch prototype. It consists of two nodes, mediation batch and rating batch, each implemented as a separate spring batch application. The nodes are integrated using Apache Camel*Apache Camel* (2014), an Java integration framework based on enterprise integration patterns, as descriped by Hohpe et al.Hohpe and Woolf (2003) Apache Camel is responsible for listening on the file system, calling the Spring batch application when a file arrives and transferring the output from the mediation batch node to the rating batch node using ftp.

The batch prototype performs the following steps:

1. The *Event generator* generates call detail records and writes them to a single file.

2. The *Mediation component* opens the file, processes it and writes the output to a single output file. The output file is getting transfered using FTP to the *Rating component*.

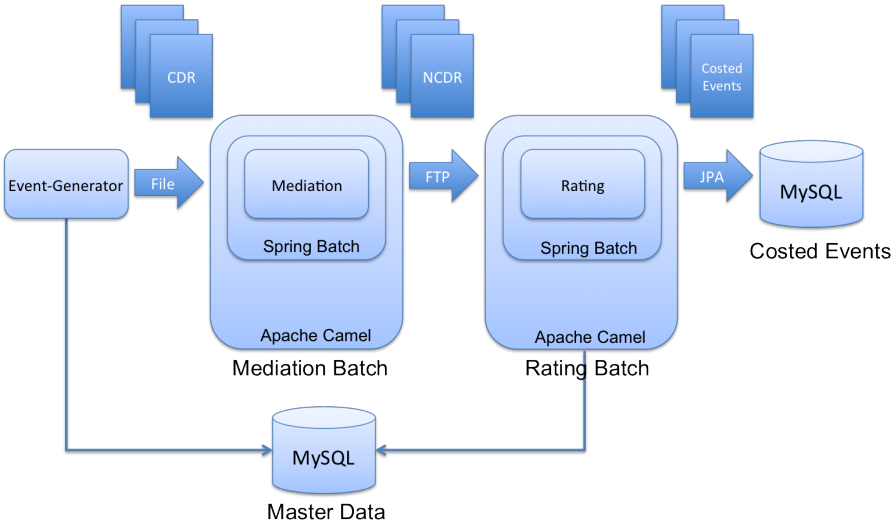


Figure 10: Batch prototype

3. The *Rating component* opens the file, processes it and writes the costed events to the costed event database.

4.1.2 *Messaging prototype*

The messaging prototype implements the billing prototype utilizing the message-oriented processing type. It uses Apache Camel*Apache Camel* (2014) as the messaging middleware.

Figure 32 shows the architecture of the messaging prototype. It consists of three nodes, the billing route, mediation service and rating service. The billing route implements the main flow of the application. It is responsible for reading messages from the billing queue, extracting the payload, calling the mediation and rating service and writing the processed messages to the database. The mediation service is a web-service representing the mediation component. It is a SOAP service implemented using Apache CXF and runs inside an Apache Tomcat container. The same applies to the rating service, representing the rating component.

The messaging prototype performs the following steps:

1. The message is read from the billing queue using JMS. The queue ist hosted by an Apache ActiveMQ instance.
2. The message is unmarshalled using JAXB.
3. The *Mediation service* is called by the CXF Endpoint of the billing route.
4. The response of the *Mediation webservice*, the normalized call detail record, is unmarshalled.

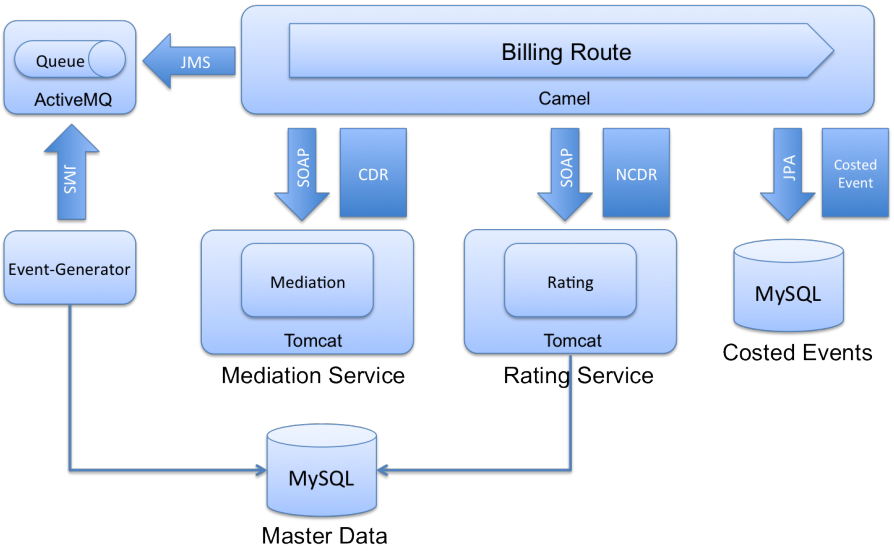


Figure 11: Message-based prototype

5. The *Rating service* is called by the CXF Endpoint of the billing route.
6. The response of the *Rating webservice*, that is the costed event, is unmarshalled.
7. The costed event is written to the *Costed Events* DB.

4.2 PERFORMANCE EVALUATION

We have conducted a performance evaluation to compare the performance characterics of the two processing types, batch processing and message-based processing, with the main focus on latency and throughput.

4.2.1 *Measuring points*

A number of measuring points have been defined for each prototype by breaking down the processing in single steps and assigning a measuring point to each step. Figure 12 and 13 show the measuring points of the batch prototype and the messaging prototype.

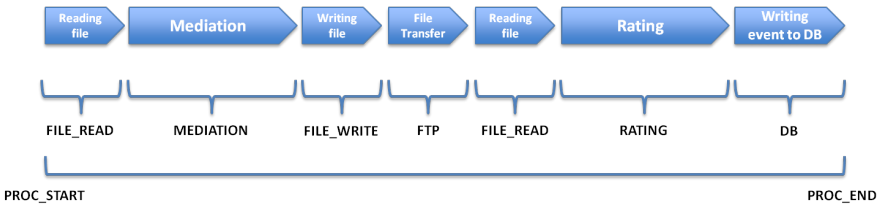


Figure 12: Measuring points of the batch prototye

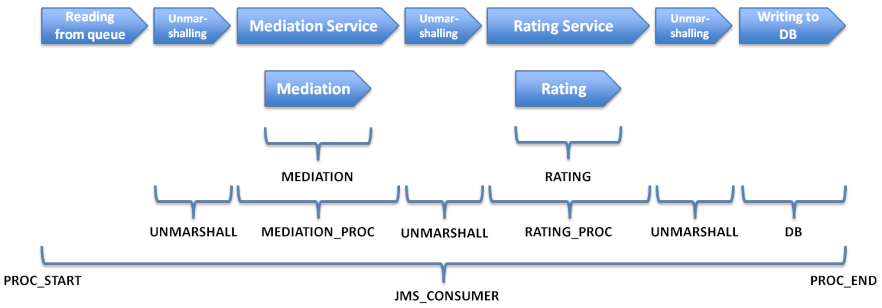


Figure 13: Measuring points of the messaging prototype

Table 2: Measuring points of the batch prototype

Measuring point	**Description**
PROC_START	Timestamp denoting the start of processing an event
PROC_END	Timestamp denoting the end of processing an event
FILE_READ	Elapsed time for reading events from file
MEDIATION	Elapsed time used by the mediation component
FILE_WRITE	Elapsed time for writing events to file
FTP	Elapsed time for file transfer using FTP
RATING	Elapsed time used by the rating component
DB	Elapsed time for writing event to the database

A detailed description of each point is shown in Table 2 and 3.

4.2.2 *Instrumentation*

A logging statement for each measuring point has been added at the appropriate code location of the prototypes using different techniques.

1. **Directly in the code**
   Whenever possible, the logging statements have been inserted directly in the code. This has been the case, when the code that should be measured, has been written exclusively for the prototype, for example the mediation and rating components.

Table 3: Measuring points of the messaging prototype

Measuring point	**Description**
PROC_START	Timestamp denoting the start of processing an event
PROC_END	Timestamp denoting the end of processing an event
JMS_CONSUMER	Elapsed time processing a single event
UNMARSHALL	Elapsed time for unmarshalling an event
MEDIATION_PROC	Elapsed time needed for calling the mediation service
MEDIATION	Elapsed time used by the mediation component
RATING_PROC	Elapsed time needed for calling the rating service
RATING	Elapsed time used by the rating component
DB	Elapsed time for writing event to the database

2. **Delegation**
   When the code to instrument has been part of a framework that is configurable using Spring, an instrumented delegate has been used.

3. **AOP**
   Finally, when the code that should get instrumented was part of a framework that was not configurable using Spring, the logging statements have been added using aspects, which are woven into the resulting class files using AspectJ.

4.2.3 *Test environment*

The two prototypes have been deployed to an Amazon EC2 environment to conduct the performance evaluation, with the characterics described in Table 4.

The batch prototype comprises two EC2 nodes, the *Mediation Node* and the *Rating Node*, containing the *Mediation Batch* and the *Rating Batch*, respectively. The *Costed Event Database* is hosted on the *Rating Node* as well. Figure 14 shows the deployment diagramm of the Batch prototype.

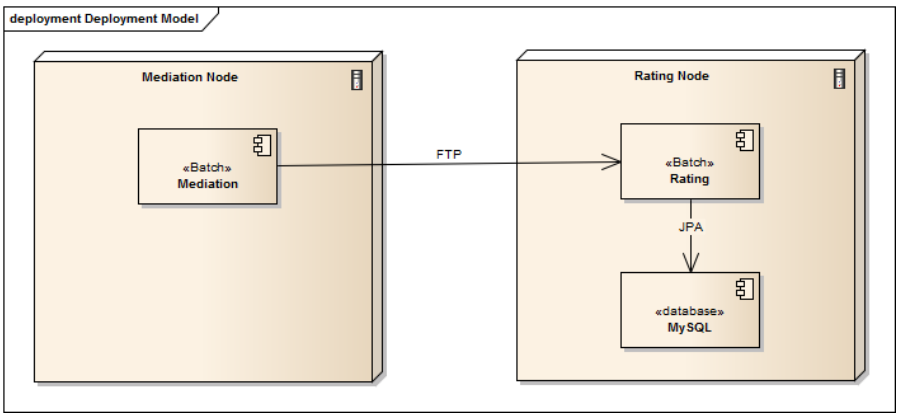


Figure 14: Batch prototype deployment on EC2 instances

The messaging prototype consists of three EC2 nodes, as shown in Figure 15. The *Master Node* hosts the *ActiveMQ Server* which runs the JMS queue containining the billing events, the *Billing Route*, which implements the processing flow of the prototype and the *MySQL Database* containing the *Costed Event Database*. The *Mediation Node* and *Rating Node* are containing the *Mediation Service* and *Rating Service*, respectively, with each service running inside an Apache Tomcat container.

The clocks of the *Mediation Node* and *Rating Node* are synchronized with the clock of the *Master Node* using PTPd*PTP daemon (PTPd)* (2013), an implementation of the Precision Time ProtocolIEEE (2008). The clock of the *Master Node* itself is synchronised with a public time-

server using the Network Time Protocol (NTP). Using this approach, a sub-millisecond precision is achieved.

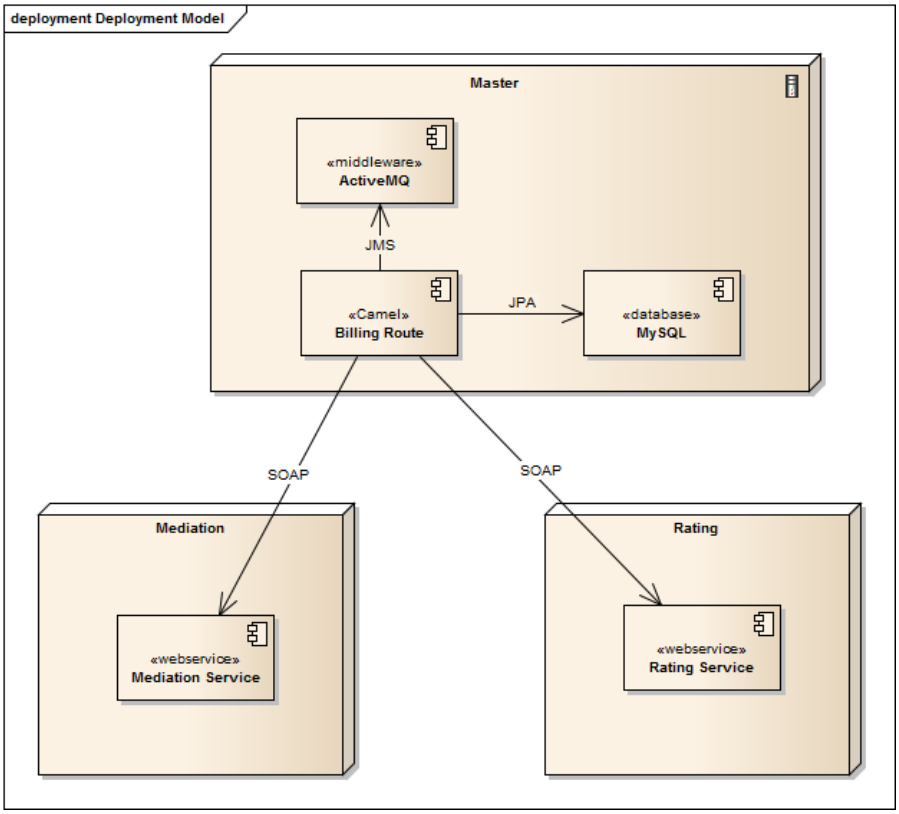


Figure 15: Messaging prototype deployment on EC2 instances

4.2.4 *Preparation and execution of the performance tests*

For running the performance tests, the Master Data DB has been set up with a list of customers, accounts, products and tariffs with each prototype using the same database and data. While part of the test-data like the products and tariffs have been created manually, the relationship between the customers and the products have been generated by a test data generator.

After setting up the master data, a number of test runs have been executed using different sizes of test data (1.000, 5.000, 10.000, 50.000,100.000, 500.000, 1.000.000 records). To get reliable results, each test configuration has been run three times. Out of the three runs for each configuration, the run having the median processing time has been used for the evaluation.

For each test run, the following steps have been executed:

1. **Generating test data**
   In case of the batch prototype, the event generator writes the test data to file. In case of the messaging prototype, the event generator writes the test data to a JMS queue.

Table 4: Amazon EC2 instance configuration

Instance type	M1 Extra Large (EBS optimized)
Memory	15 GiB
Virtual Cores	8 (4 cores x 2 units)
Architecture	64-bit
EBS Volume	10 GiB (100 IOPS)
Instance Store Volumes	1690 GB (4x420 GB Raid 0)
Operating System	Ubuntu 12.04 LTS (GNU/Linux 3.2.0-25-virtual x86_64)
Database	MySQL 5.5.24
Messaging Middleware	Apache ActiveMQ 5.6.0

2. **Running the test**
   Each prototype listens on the file system and the JMS queue, respectively. Using the batch prototype, the processing starts when the input file is copied to the input folder of the mediation batch application by the event genarotor. Using the messaging prototype, the processing starts when the first event is written to the JMS queue by the test generator.

3. **Validating the results**
   Processsing the log files written during the test run

4. **Cleaning up**
   Deleting the created costed events from the DB.

Before running the tests, each prototype has been warmed up by processing 10.000 records.

4.2.5 *Results*

The performance evaluation yields the following results.

4.2.5.1 *Throughput*

The throughput per second for a test run with N records is defined as

$$TP/s_N = N/PT_N$$

with PT_N being the total processing time for N records. Figure 16 shows the measured throughput of the batch and messaging prototypes. The messaging prototype is able to process about 70 events per second. The maximum throughput of the batch prototype is about 383 records per second which is reached with an input of 1.000.000 records.

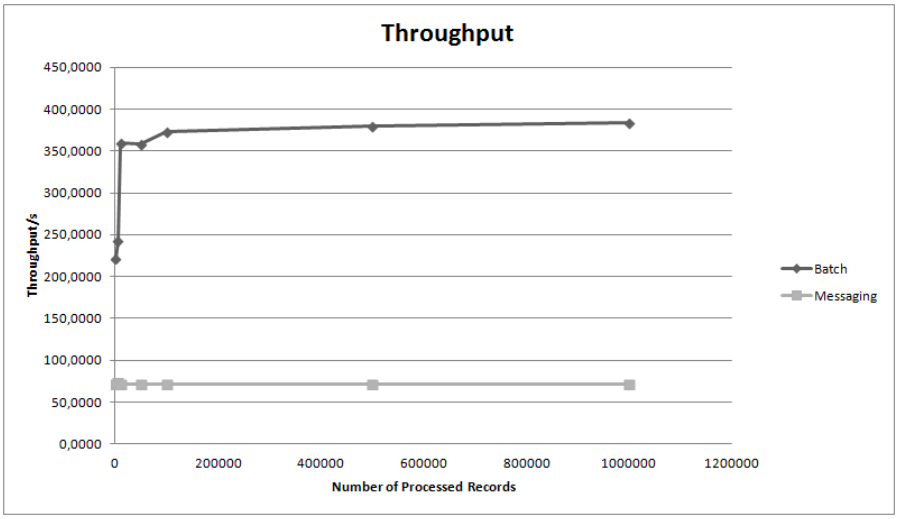


Figure 16: Throughput

4.2.5.2 *Latency*

Figure 17 shows the measured latencies of the batch and messaging prototypes. To rule out peaks, the 95th percentile has been used, that is, 95% of the measured latencies are below this value. In case of the batch prototype, the 95th percentile latency is a linear function of the amount of data. The latency increases proportionally to the number of processd records. In case of the messaging prototype, the 95th percentile latency is approximately a constant value which is independant of the number of processed records.

4.2.5.3 *Processing overhead*

The overhead of the batch prototype is about 7% of the total processing time, independant of the number of processed records, as shown in Figure 18. This overhead contains file operations, such as opening, reading, writing and closing of input files, the file transfer between

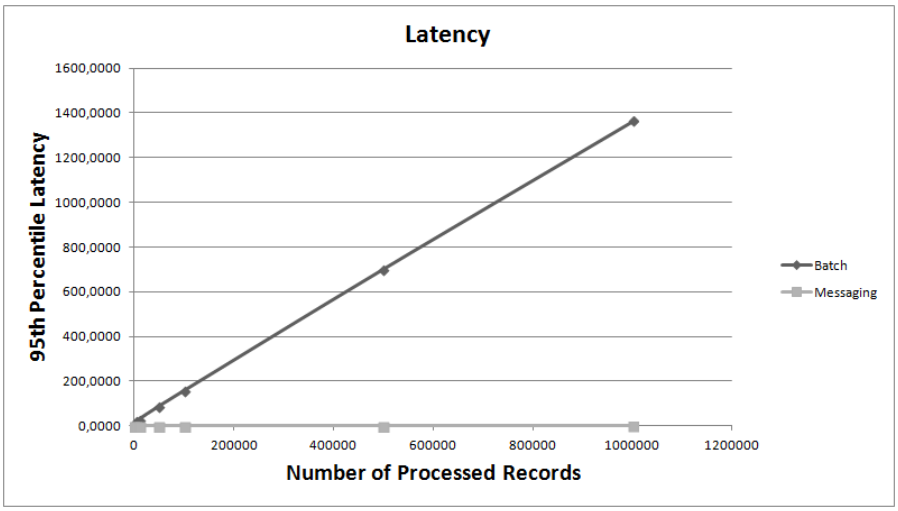


Figure 17: Latency

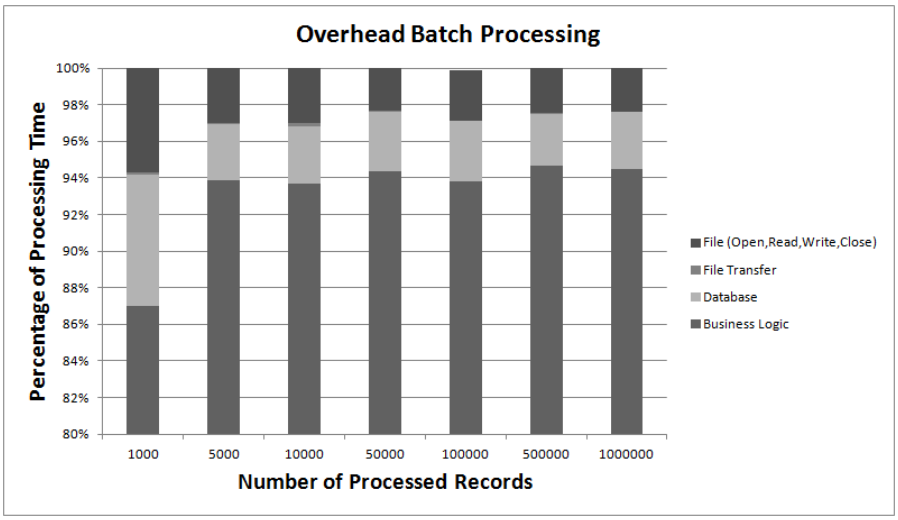


Figure 18: Overhead batch prototype

the Mediation and Rating Nodes and the database transactions to write the the processed event to the Costed Events DB.

On the contrary, the overhead of the messaging prototype is about 84% of the total processing time (see Figure 19). In case of the messaging prototype, the overhead contains the JMS overhead, that is the overhead for reading events from the message queue, the webservice overhead needed for calling the Mediation and Rating services including marshalling and unmarshalling of input data and the overhead caused the database transactions to write the processed events to the Costed Events DB. Most of the overhead is induced by the webservice overhead and the database overhead. Since every event is written to the database in its own transaction, the database overhead of the messaging prototype is much larger than the database overhead of the batch prototype.

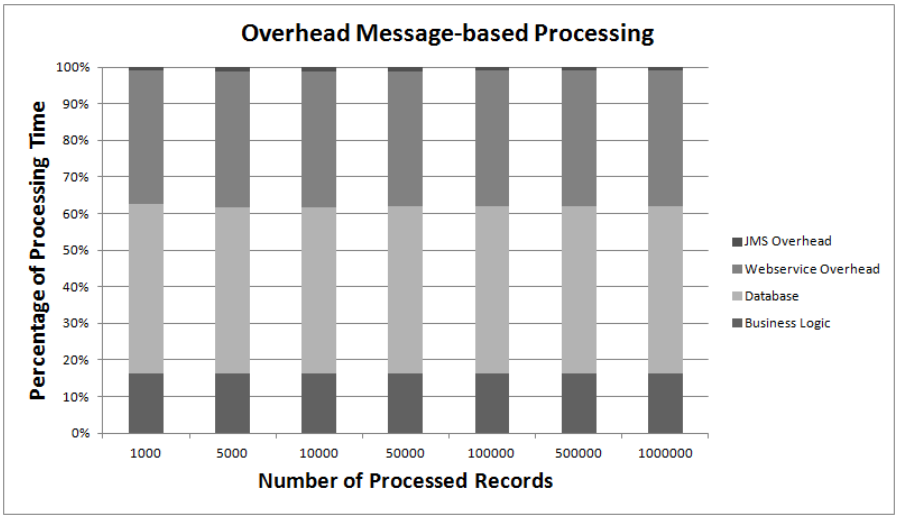


Figure 19: Overhead messaging prototype

4.2.5.4 *System utilisation*

The system utilisation has been measured using the sar (System Activity Report) command while running the performance tests. Figure 20 shows the mean percentage of CPU consumption at the user level (%user) and the mean percentage of used memory (%memused) for the Mediation node and Rating node of the Batch prototype. The CPU utilisation of Medation Node and Ratig Node is about 2% and 19%, respectively. The memory utilisation increases slowly with the number of processed records.

Figure 21 shows the mean CPU consumption and mean memory usage for the nodes of the Messaging prototype. The CPU utilisation of the Master Node, Mediation Node and Rating Node is about 9%, 1% and 6%, respectively. As the same with the batch prototye, the memory utilisation of the messaging prototype increases with

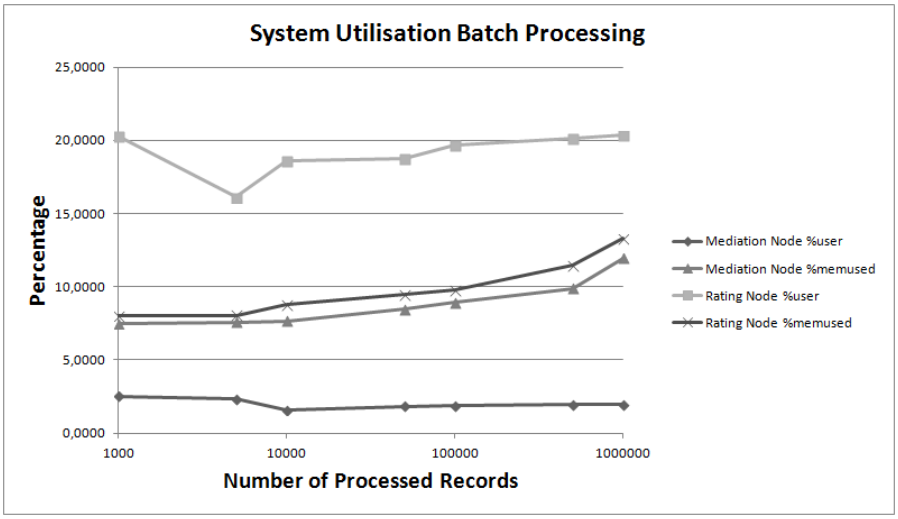


Figure 20: System utilisation batch prototype

the number of processed records. The memory utilisation of the master node peaks at about 38% with 500000 processed records. With 1000000 processed records, the memory utilisation is only about 25%, which presumably can be accounted to the garbage collector.

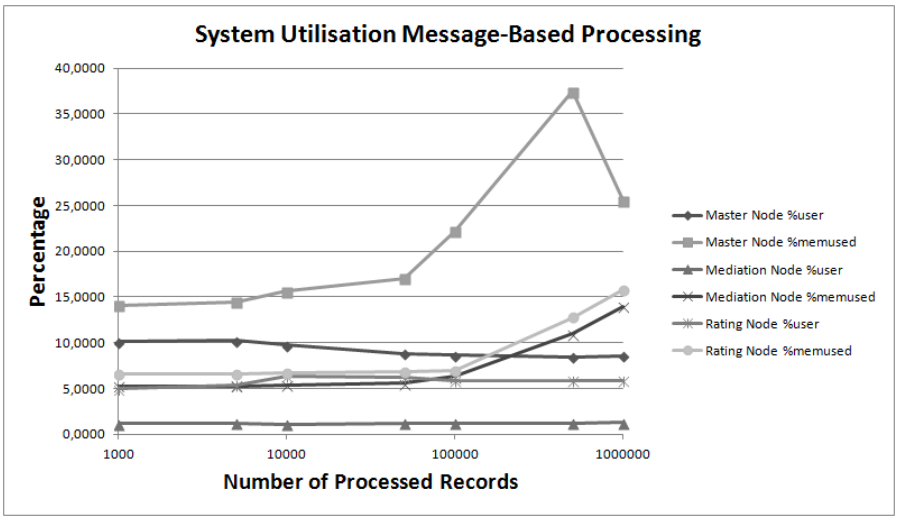


Figure 21: System utilisation messaging prototype

4.3 IMPACT OF DATA GRANULARITY ON THROUGHPUT AND LATENCY

The results presented in Section 4.2.5 suggest that the throughput of the messaging prototype can be increased by increasing the granularity of the data that is beeing processed. Data granularity relates to the amount of data that is processed in a unit of work, for example in a single batch run or an event. In order to examine this approach, we

have repeated the performance tests using different package sizes for processing the data.

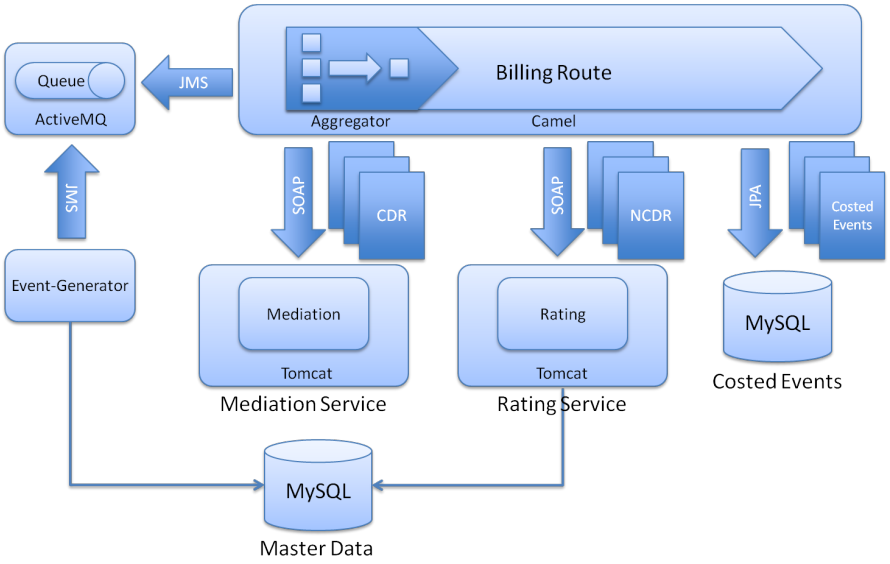


Figure 22: The data granularity is controlled by an aggregator

For this purpose, the messaging prototype has been extended to use an aggregator in the messaging route. The aggregator is a stateful filter which stores correlated messages until a set of messages is complete and sends this set to the next processing stage in the messaging route. In case of the messaging prototype, messages are not corelated to each other and also the messages can be processed in an arbitrary order. A set of messages is complete when it reaches the configured package size. In other scenarios, it is possible to corelate messages by specific data, for example an account number or by a business rule.

Figure 33 shows the impact of different aggregatation sizes on the throughput of the messaging prototype. For each test 100.000 events have been processed. The throughput increases constantly for $1 < aggregation\_size <= 50$ with a maximum of 673 events per second with $aggregation\_size = 50$. Higher aggregation sizes than 50 do not further increase the throughput, it stays around 390 events per second. Surprisingly, the maximum throughput of 673 events per second even outperforms the throughput of the batch prototype which is about 383 records per second. This is presumably a result of the better multithreading capabilities of the camel framework.

Increasing the aggregation size also decreases the processing overhead, as shown in Figure 24. An aggregate size of 10 decreases the overhead by more than 50% compared to an aggregate size of 1. Of course, the integration of the aggregator adds an additional overhead which is insignificant for $aggregation\_size > 50$.

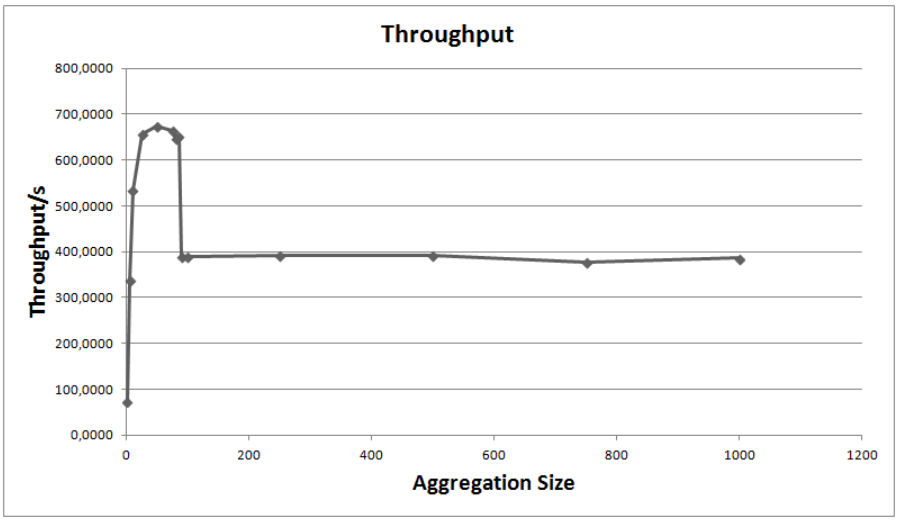


Figure 23: Impact of different aggregation sizes on throughput

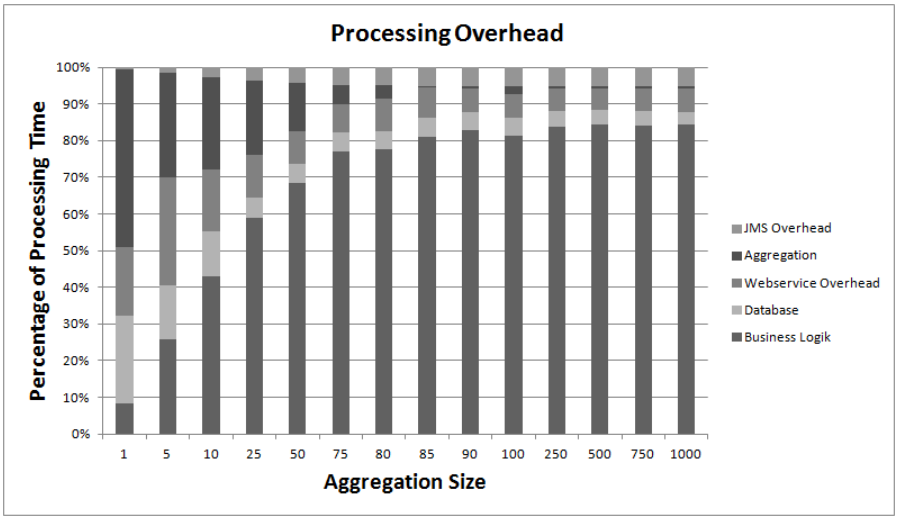


Figure 24: Impact of different aggregation sizes on processing overhead

The increased throughput achieved by increasing the aggregation size comes with the cost of a higher latency. Figure 34 shows the impact of different aggregation sizes on the 95th percentile latency of the messaging prototype.

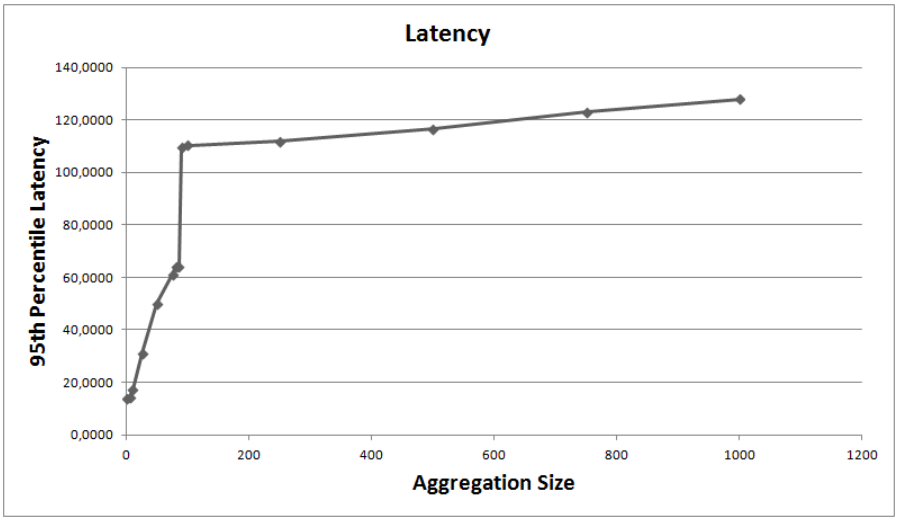


Figure 25: Impact of different aggregation sizes on latency

An aggregation size of 50, resulting in the maximum throughput of 673 events per seconds, shows a 95th percentile latency of about 68 seconds. This latency is significantly higher than the latency of the messaging system without message aggregation, which is about 0,15 seconds (see Section 4.2.5.2).

Figure 26 shows the impact of different aggregation sizes on the system utilisation. The CPU utilisation of the Master node shows a maximum of 30% with an aggregation size of 25. An $aggregation\_size >= 90$ results in a CPU utilisation of about 15%. The maximum memory utilisation of the Master node is 41% with an aggregation size of 100.

The maximum system utilisation of the Rating node is 25% with an aggregation size of 80. The memory utilisation is between 7-8% irrespective of aggregation size. Maximum system and memory utilisation of the Mediation node are also irrespective of aggregation size, beeing less than 2% and 8%, respectively.

When using high levels of data granularity, the messaging system is essentially a batch processing system, providing high throughput with high latency. To provide near-time processing an optimum level of data granularity would allow having the lowest possible latency with the lowest acceptable throughput.

4.4 RELATED WORK

This section gives an overview of work related to the research presented in this paper. Related work can be categorised in two different topics, performance measuring and performance optimisation of

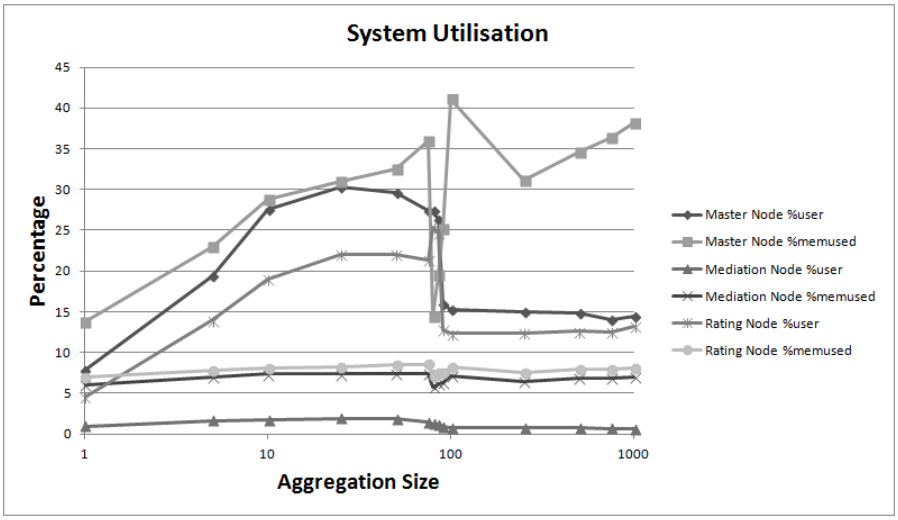


Figure 26: Impact of different aggregation sizes on system utilisation

messaging systems. Performance measuring is applied to evaluate if an implemented system meets its performance requirements and to spot possible performance problems. Performance optimisation aims to improve the performance of a system to meet certain requirements such as throughput or latency.

4.4.1 *Performance Measuring*

Her et al. Her et al. (2007) propose the following set of metrics for measuring the performance of a service-oriented system:

- **Service response time**
  Elapsed time between the end of request to service and the beginning of the response of the service. This metric is further split in 20 sub-metrics such as message processing time, service composition time and service discovery time.

- **Think time**
  Elapsed time between the end of a response generated by a service and the beginning of a response of an end user.

- **Service tournaround time**
  Time needed to get the result from a group of related activities within a transaction.

- **Throughput**
  Number of requests served at a given period of time. The authors distinguish between the throughput of a service and the throughput of a business process.

In their work, Henjes et al. Henjes et al. (2006)Menth et al. (2006) investigated the throughput performance of the JMS server FioranaMQ,

SunMQ and WebsphereMQ. The authors came to the following conclusion:

- Message persistence reduces the throughput significantly.
- Message replication increases the overall throughput of the server.
- Throughput is limited either by the processing logic for small messages or by the transmission capacity for large messages.
- Filtering reduces the throughput significantly.

Chen and Greenfield Chen and Greenfield (2004) propose that the following performance metrics should be used to evaluate a JMS server:

- Maximum sustainable throughput
- Latency
- Elapsed time taken to send batches messages
- Persistent message loss after recovery

The authors state that "although messaging latency is easy to understand, it is difficult to measure precisely in a distributed environment without synchronised high- precision clocks." They discovered that latencies increase with increasing message sizes.

SPECjms2007 is a standard benchmark for the evaluation of Message-Oriented Middleware platforms using JMS Sachs et al. (2009). It provides a flexible performance analysis framework for tailoring the workload to specific user requirements. According to Sachs et al. Sachs et al. (2007), the workload of the SPECjms2007 benchmark has to meet the following requirements:

- **Representativeness**
  The workload should reflect how the messaging platform is used in typical user scenarios.
- **Comprehensiveness**
  The workload should incorporate all platform features typically used in JMS application including publish/subscript and point-to-point messaging.
- **Focus**
  The workload should focus on measuring the performance of the messaging middleware and should minimize the impact of other components and services.
- **Configurability**
  It should be possible to configure the workload to meet the requirements of the user.

- **Scalability**
  It should be possible to scale the workload by the number of destinations with a fixed traffic per destination or by increasing the traffic with a fixed set of destinations.

Ueno and Tatsubori propose a methodology to evaluate the performance of an ESB in an early stage of development that can be used for capacity planning Ueno and Tatsubori (2006). Instead of using a performance model for performance prediction, they run the ESB on a real machine with a pseudo-environment using lightweight web service providers and clients. The authors state that model-based approaches "often require elemental performance measurements and sophisticated modeling of the entire system, which is usuable not feasible for complex systems".

4.4.2 *Performance Optimisation*

Various approaches have been proposed to optimise the performance of webservices, in particular SOAP, the standard protocol for Web Service communication. This includes approaches for optimising the processing of SOAP messages (see for example Abu-Ghazaleh and Lewis (2005), Suzumura et al. (2005) and Ng (2006)), compression of SOAP messages (see for example Estrella et al. (2008) and Ng et al. (2005)) and caching (see for example Andresen et al. (2004) and Devaram and Andresen (2003)). A survey of the current approaches to improve the performance of SOAP can be found in Tekli et al. (2012).

Wichaiwong and Jaruskulchai Wichaiwong and Jaruskulchai (2007) propose an approach to transfer bulk data between web services per FTP. The SOAP messages transferred between the web services would only contain the necessary details how to download the corresponding data from an FTP server since this protocol is optimized for transferring huge files. This approach solves the technical aspect of efficiently transferring the input and output data but does not pose any solutions how to implement loose coupling and how to integrate heterogeneous technologies, the fundamental means of an SOA to improve the flexibility of an application landscape.

Data-Grey-Box Web Services are an approach to transfer bulk data between Web Services Habich, Richly and Grasselt (2007). Instead of transferring the data wrapped in SOAP messages, it is transferred using an external data layer. For example when using database systems as data layer, this facilitates the use of special data transfer methods such ETL (Extract, Transform, Load) to transport the data between the database of the service requestor and the database of the Web service. The data transfer is transparent for both service participants in this case. The approach includes an extension of the Web service interface with properties describing the data aspects. Compared to the SOAP approach, the authors measured a speedup of up to 16 using

their proposed approach. To allow the composition and execution of Data-Grey-Box Web services, Habich et al. Habich, Richly, Preissler, Grasselt, Lehner and Maier (2007) developed BPEL data transitions to explicitly specify data flows in BPEL processes.

Zhuang and Chen propose three tuning strategies to improve the performance of Java Messaging (JMS) for cloud-based applications Zhuang and Chen (2012).

1. When using persistent mode for reliable messaging the storage block size should be matched with the message size to maximise message throughput.

2. Applying distributed persistent stores by configuring multiple JMS destinations to achieve parallel processing

3. Choosing appropriate storage profiles such as RAID-1

MPAB (Massively Parallel Application Bus) is an ESB-oriented messaging bus used for the integration of business applications Benosman et al. (2012). The main principle of MPAB is to fragment an application into parallel software processing units, called SPU. Every SPU is connected to an Application Bus Multiplexor (ABM) through an interface called Application Bus Terminal (ABT). The Application Bus Multiplexor manages the resources shared across the host system and communicates with other ABM using TCP/IP. The Application Bus Terminal contains all the resources needed by SPU to communicate with its ABM. A performance evaluation of MPAB shows that it achieves a lower response time compared to the open source ESBs Fuse, Mule and Petals.

Some research has been done to add real-time capabalities to ESB or messaging middleware. Garces-Erice proposes an architecture for a real-time messaging middleware based on an Enterprise Service Bus Garces-Erice (2009). It consists of an event scheduler, a JMS-like API and a communication subsystem. While fulfilling real-time requirements, the middleware also supports already deployed infrastructure.

In their paper, Xia and Song suggest a real-time ESB model by extending the JBI specification with semantics for priority and time restrictions and modules for flow control and bandwith allocation. The proposed system is able to dynamically allocate bandwidth according to business requirements.

Tempo is a real-time messaging system written in Java that can be used on either a real-time or non-real-time architecture Bauer et al. (2008). The authors, Bauer et al., state that existing messaging systems are designed for transactional processing and therefore not appropriate for applications with with stringent requirements of low latency with high throughtput. The main principle of Tempo is to use

an independent queuing system for each topic. Ressources are partitioned between these queueing systems by a messaging scheduler using a time-base credit scheduling mechanism. In a test environment, Tempo is able to process more than 100.000 messages per second with a maximum latency of less than 120 milliseconds.

Haesen et al. distinguishes between two types of data granularity Haesen et al. (2008):

- **Input data granularity**
  Data that is sent to a component
- **Output data granularity**
  Data that is returned by a component

The authors state that a coarse-grained data granularity reduces the communication overhead, since the number of network transfers is decreased. "Especially in the case of Web services, this overhead is high since asynchronous messaging requires multiple queuing operations and numerous XML transformations".

4.5 SUMMARY

Near-time processing of bulk data is hard to achieve. As shown in Section 5.2.3, latency and throughput are opposed performance metrics of a system for bulk data processing. High throughput, as provided by batch processing, leads to high latency, which impedes near-time processing. On the other hand, low latency, as provided by a message-based system, cannot provide the throughput needed for bulk data processing due the additional overhead for each processed message.

While it is technically possible to minimise the overhead of a message-based system by implementing a lightweight marshalling system and not use JMS or other state-of-the-art technologies such as XML, SOAP or REST, it would hurt the ability of the messaging middleware to integrate heterogenous systems or services and thus limiting its flexibility, which is one the main selling propositions of such a middleware. Furthermore, batch processing enables optimizations by partitioning and sorting the data appropriately which is not possible when each record is processed independently as a single message.

In order to compare throughput and latency of batch and message-oriented systems, a prototype for each processing type has been built. A performance evaluation has been conducted with the following results:

- The throughput of the batch prototype is 4 times the throughput of the messaging prototype.
- The latency of the messaging prototype is only a fraction of the latency of the batch prototype.

- The overhead of the messaging prototype is about 84% of the total processing time, which is mostly induced by the webservice overhead and the database transactions.
- The overhead of the batch prototype is only about 7% of the total processing time.

The results presented in Section 4.3 show that throughput and latency depend on the granularity of data that is being processed. The throughput of the messaging-prototype can be increased by aggregating messages with the cost of a higher latency. An optimum data granularity would allow having the lowest possible latency with the lowest acceptable throughput and thus providing near-time processing of bulk data.

To achieve the lowest possible latency while still providing the lowest acceptable maximum throughput of the system, the granularity of the data processed in one message could be adjusted at runtime by a middleware service which constantly measures the throughput and latency of the system and controls the granularity of the data. If the throughput drops below the acceptable minimum, the granularity of the data needs to be higher. On the other hand, the granularity can be lowered, if the throughput of the system is above the minimum. The next part of this research will implement and evaluate such a middleware service for messaging systems.

5

AN ADAPTIVE MIDDLEWARE FOR NEAR-TIME PROCESSING OF BULK DATA

5.1 INTRODUCTION

Enterprise Systems like customer-billing systems or financial transaction systems are required to process large volumes of data in a fixed period of time. For example, a billing system for a large telecommunication provider has to process more than 1 million bills per day. Those systems are increasingly required to also provide near-time processing of data to support new service offerings.

Traditionally, enterprise systems for bulk data processing are implemented as batch processing systems (Fleck, 1999). Batch processing delivers high throughput but cannot provide near-time processing of data, that is the end-to-end latency of such a system is high. End-to-end latency refers to the period of time that it takes for a business process, implemented by multiple subsystems, to process a single business event. For example, consider the following billing system of telecommunications provider:

- Customers are billed once per month
- Customers are partitioned in 30 billing groups
- The billing system processes 1 billing group per day, running 24h under full load.

In this case, the mean time for a call event to be billed by the billing system is 1/2 month. That is, the mean end-to-end latency of this system is 1/2 month.

A lower end-to-end latency can be achieved by using single-event processing, for example by utilizing a message-oriented middleware for the integration of the services that form the enterprise system. While this approach is able to deliver near-time processing, it is hardly capable for bulk data processing due to the additional communication overhead for each processed message. Therefore, message-based processing is usually not considered for building a system for bulk data processing requiring high throughput.

The processing type is usually a fixed property of an enterprise system that is decided when the architecture of the system is designed, prior to implementing the system. This choice depends on the non-functional requirements of the system. These requirements are not fixed and can change during the lifespan of a system, either anticipated or not anticipated.

Additionally, enterprise systems often need to handle load peaks that occur infrequently. For example, think of a billing system with moderate load over most of the time, but there are certain events with very high load such as New Year's Eve. Most of the time, a low end-to-end latency of the system is preferable when the system faces moderate load. During the peak load, it is more important that the system can handle the load at all. A low end-to-end latency is not as important as an optimized maximum throughput in this situation.

In this paper, we propose a solution to this problem:

- We introduce the concept of a middleware that is able to adapt its processing type fluently between batch processing and single-event processing. By adjusting the data granularity at runtime, the system is able to minimize the end-to-end latency for different load scenarios. (Section 5.3)

The remainder of this paper is organized as follows. Section 5.2 defines the considered type of system and the terms throughput and latency. The proposed middleware and the results of preliminary performance tests are presented in Section 5.3. Section 5.4 gives an overview of other work related to this reasearch. Finally, Section 5.5 concludes the paper and gives and outlook to the next steps of this research.

5.2 BACKGROUND

We consider a distributed system for bulk data processing consisting of several subsystems running on different nodes that together form a processing chain, that is, the output of subsystem S1 is the input of the next subsystem S2 and so on (see Figure 27a).

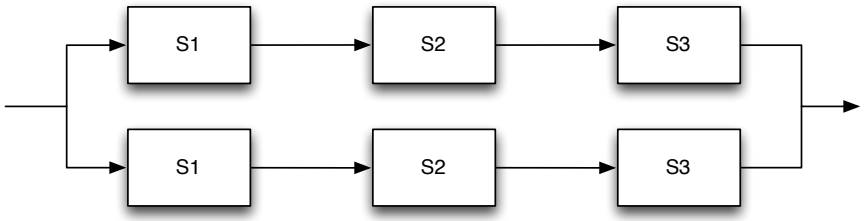
To facilitate parallel processing, the system can consist of several lines of subsystems with data beeing distributed among each line. For simplification, we consider a system with a single processing line in the remainder of this paper.

We discuss two processing types for this kind of system, batch processing and message-based processing.

5.2.1 *Batch processing*

The traditional operation paradigm of a system for bulk data processing is batch processing (see Figure 28). A batch processing system is an application that processes bulk data without user interaction. Input and output data is usually organized in records using a file- or database-based interface. In the case of a file-based interface, the application reads a record from the input file, processes it and writes the record to the output file.

(a) Single processing line



(b) Parallel processing lines

Figure 27: A system consisting of several subsystems forming a processing chain

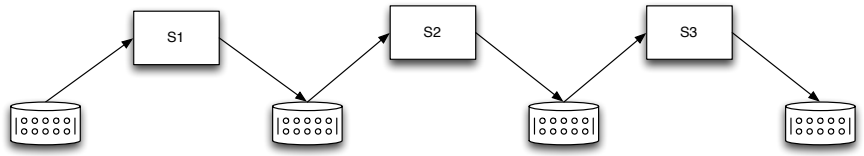


Figure 28: Batch processing

5.2.2 *Message-base processing*

Messaging facilitates the integration of heterogeneous applications using asynchronous communication. Applications are communicating with each other by sending messages (see Figure 29). A messaging server or message-oriented middleware handles the asynchronous exchange of messages including an appropriate transaction control (Conrad et al., 2006).

Figure 29: Message-based processing

Message-based systems are able to provide near-time processing of data due to their lower latency compared with batch processing systems. The advantage of a lower latency comes with a performance cost in regard to a lower maximum throughput because of the additional overhead for each processed message. Every message needs, amongst others, to be serialized and deserialized, mapped between different protocols and routed to the appropriate receiving system.

5.2.3 *End-to-end Latency vs. Maximum Throughput*

Throughput and latency are performance metrics of a system. We are using the following definitions of maximum throughput and latency in this paper:

- **Maximum Throughput**
  The number of events the system is able to process in a fixed timeframe.
- **End-To-End Latency**
  The period of time between the occurrence of an event and its processing. End-to-end latency refers to the total latency of a complete business process implemented by multiple subsystems. The remainder of this paper focusses on end-to-end latency using the general term latency as an abbreviation.

Latency and maximum throughput are opposed to each other given a fixed amount of processing resources. High maximum throughput, as provided by batch processing, leads to high latency, which impedes near-time processing. On the other hand, low latency, as provided by a message-based system, cannot provide the maximum throughput needed for bulk data processing because of the additional overhead for each processed event.

5.3 AN ADAPTIVE MIDDLEWARE FOR NEAR-TIME PROCESSING OF BULK DATA

This section introduces the concept of an adaptive middleware which is able to adapt its processing type fluently between batch processing and single-event processing. It continuously monitors the load of the system and controls the message aggregation size. Depending on the current aggregation size, the middleware automatically chooses the appropriate service implementation and transport mechanism to further optimize the processing.

5.3.1 *Middleware Components*

Figure 30 shows the components of the middleware, that are based on the Enterprise Integration Patterns described by Hohpe and Woolf (2003).

5.3.1.1 *Aggregator*

The Aggregator is a stateful filter which stores correlated messages until a set of messages is complete and sends this set to the next processing stage in the messaging route.

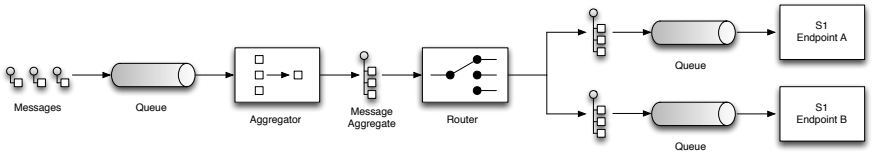


Figure 30: Components of the Adaptive Middleware. We are using the notation defined by Hohpe and Woolf (2003)

There are different options to aggregate messages, which can be implemented by the Aggregator:

- **No correlation**: Messages are aggregated in the order in which they are read from the input message queue. In this case, an optimized processing is not simply possible.
- **Technical correlation:** Messages are aggregated by their technical properties, for example by message size or message format.
- **Business correlation**: Messages are aggregated by business rules, for example by customer segments or product segments.

5.3.1.2 *Feedback Loop*

To control the level of message aggregation at runtime, the middleware uses a closed feedback loop with the following properties (see Figure 31):

- **Input (u):** Current aggregation size
- **Output (y):** Change of queue size measured between sampling intervals
- **Set point (r):** The change of queue size should be zero.

Ultimately, we want to control the average end-to-end latency depending on the current load of the system. The change of queue size seems to be an appropriate quantity because it can be directly measured without a lag at each sampling interval, unlike the average end-to-end latency.

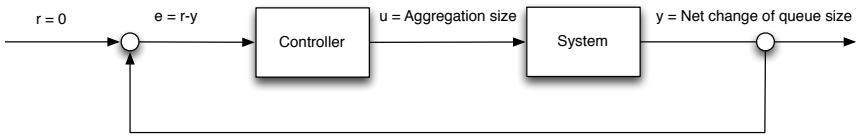


Figure 31: Feedback loop to control the aggregation size

The concrete architecture and tuning of the feedback loop and the controller is subject to our ongoing research.

5.3.1.3 *Router*

Depending on the size of the aggregated message, the Router routes the message to the appropriate service endpoint, which is either optimized for batch or single event processing.

When processing data in batches, especially when a batch contains correlated data, there are multiple ways to speed up the processing:

- To reduce I/O, data can be pre-loaded at the beginning of the batch job and held in memory.
- Storing calculated results for re-use in memory
- Use bulk database operations for reading and writing data

With high levels of message aggregation, it is not preferred to send the aggregated message payload itself over the message bus using Java Message Service (JMS) or SOAP. Instead, the message only contains a pointer to the data payload, which is transferred using File Transfer Protocol (FTP) or a shared database.

5.3.2 *Prototype Implementation*

To evaluate the proposed concepts of the adaptive middleware, we have implemented a prototype of a billing system using Apache Camel (*Apache Camel*, 2014) as the messaging middleware.

Figure 32 shows the architecture of the prototype system.

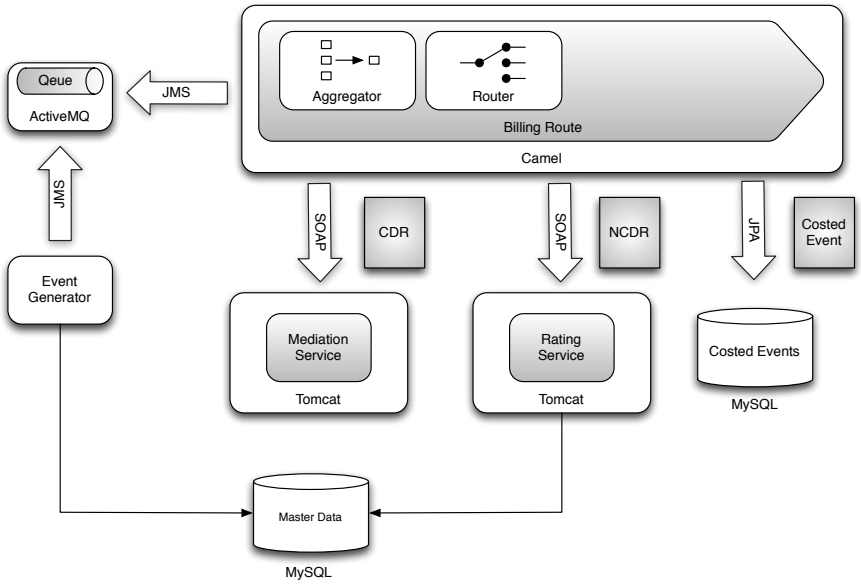


Figure 32: Architecture of the prototype system

Using this prototype, we have done some preliminary performance tests to examine the impact of message aggregation on latency and

throughput. For each test, the input message queue has been pre-filled with 100.000 events. We have measured the total processing time and the processing time of each message with different static message aggregation sizes.

Figure 33 shows the impact of different aggregation sizes on the throughput of the messaging prototype. The throughput increases

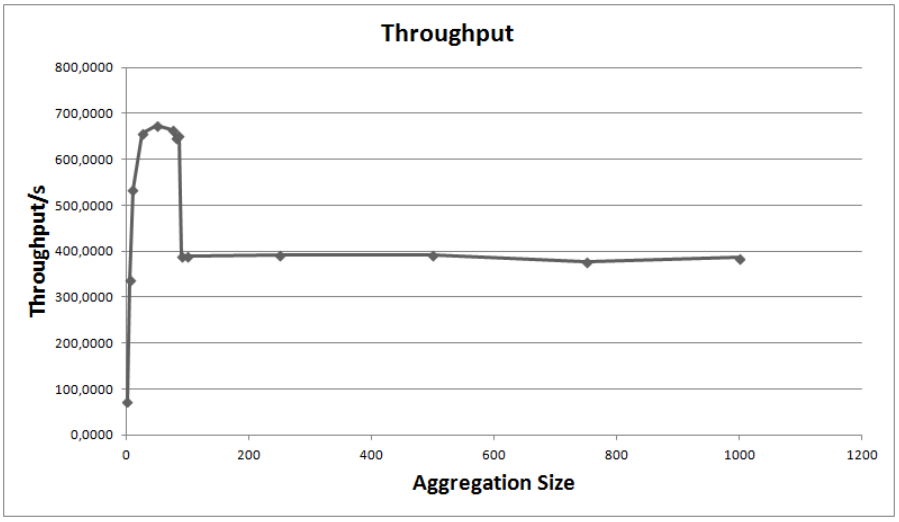


Figure 33: Impact of different aggregation sizes on throughput

constantly for $1 < aggregation\_size <= 50$ with a maximum of 673 events per second with $aggregation\_size = 50$. Higher aggregation sizes than 50 do not further increase the throughput, it stays around 390 events per second.

The increased throughput achieved by increasing the aggregation size comes with the cost of a higher latency. Figure 34 shows the impact of different aggregation sizes on the 95th percentile latency of the messaging prototype.

An aggregation size of 50, resulting in the maximum throughput of 673 events per seconds, shows a 95th percentile latency of about 68 seconds.

The results indicate that there is an optimal range for the aggregation size to control the throughput and latency of the system. Setting the aggregation size higher than a certain threshold leads to a throughput drop and latency gain. In case of our prototype, this threshold is between an aggregation size of 85 and 90. This threshold needs to be considered by the control strategy. We are currently investigating the detailed causes of this finding.

5.4 RELATED WORK

Research on messaging middleware currently focusses on Enterprise Services Bus (ESB) infrastructure. An ESB is an integration plattform that combines messaging, web services, data transformation and in-

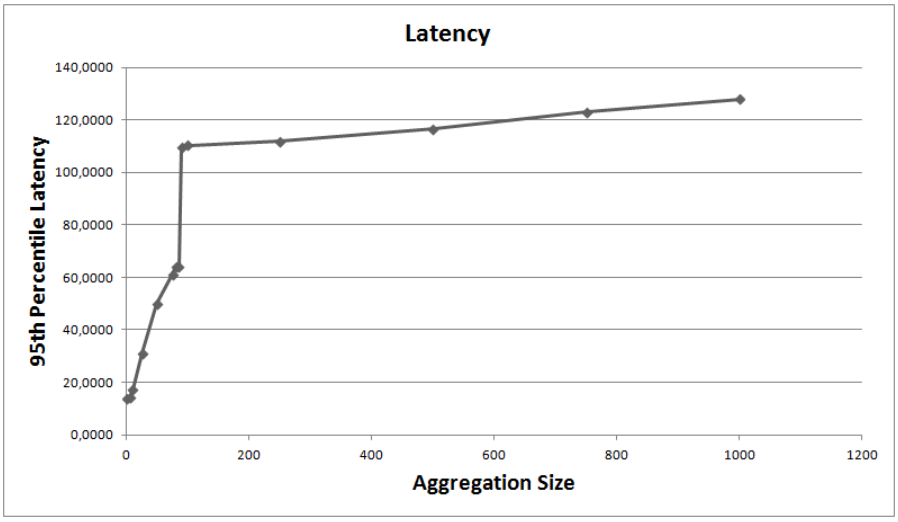


Figure 34: Impact of different aggregation sizes on latency

telligent routing to connect multiple heterogeneous services (Chappell, 2004). It is a common middleware to implement the integration layer of an Service Oriented Architecture (SOA) and is available in numerous commercial and open-source packages.

Several research has been done to extend the static service composition and routing features of standard ESB implementations with dynamic capabilities decided at run-time, such as dynamic service composition (Chang et al., 2007), routing (Bai et al., 2007) (Wu et al., 2008) (Ziyaeva et al., 2008) and load balancing (Jongtaveesataporn and Takada, 2010).

Work to manage and improve the Quality of Service (QoS) of ESB and service-based systems in general is mainly focussed on dynamic service composition and service selection based on monitored QoS metrics such as throughput, availability and response time (Calinescu et al., 2011). González and Ruggia (2011) propose an adaptive ESB infrastructure to adress QoS issues in service-based systems which provides adaption strategies for response time degradation and service saturation, such as invoking an equivalent service, using previously stored information, distributing requests to equivalent services, load balancing and deferring service requests.

The adaption strategy of our middleware is to change the message aggregation size based on the current load of the system. Aggregating or batching of messages is a common approach to increase the throughput of a messaging system, for example to increase the throughput of total ordering protocols (Friedman and Renesse, 1997) (Friedman and Hadad, 2006) (Romano and Leonetti, 2012) (Didona et al., 2012).

A different solution to handle infrequent load spikes is to automatically instantiate additional server instances, as provided by current Platform as a Service (PaaS) offerings such as Amazon EC2 (*Amazon*

EC2 Auto Scaling, n.d.) or Google App Engine (*Auto Scaling on the Google Cloud Platform*, n.d.). While scaling is a common approach to improve the performance of a system, it also leads to additional operational and possible license costs. Of course, our solution can be combined with these auto-scaling approaches.

5.5 CONCLUSION AND FUTURE WORK

In this paper, we have presented a middleware that is able to adapt itself to changing load scenarios by fluently shifting the processing type between single event and batch processing. The middleware uses a closed feedback loop to control the end-to-end latency of the system by adjusting the level of message aggregation depending on the current load of the system. Determined by the aggregation size of a messsage, the middleware routes a message to appropriate service endpoints, which are optimized for either single-event or batch processing.

To evaluate the proposed middleware concepts, we have implemented a prototype system and performed preliminary performance tests. The tests show that throughput and latency of a messaging system depend on the level of data granularity and that the throughput can be increased by increasing the granularity of the processed messages.

Next steps of our research are the implementation of the proposed middleware including the evaluation and tuning of different controller architectures, performance evaluation of the proposed middleware using the prototype and developing a conceptional framework containing guidelines and rules for the practitioner how to implement an enterprise system based on the adaptive middleware for near-time processing

6

A CONCEPTUAL FRAMEWORK FOR HIGH-PERFORMANCE NEAR-TIME PROCESSING OF BULK DATA

Part III

CONCLUSION

BIBLIOGRAPHY

Abu-Ghazaleh, N. and Lewis, M. J. (2005). Differential Deserialization for Optimized SOAP Performance, *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, IEEE Computer Society, Washington, DC, USA, p. 21.

Amazon EC2 Auto Scaling (n.d.). http://aws.amazon.com/autoscaling. [retrieved: March 2014]. (Cited on page 54.)

Andresen, D., Sexton, D., Devaram, K. and Ranganath, V. (2004). LYE: a high-performance caching SOAP implementation, *Proceedings of the 2004 International Conference on Parallel Processing (ICPP-2004)*, pp. 143–150.

Apache Camel (2014). http://camel.apache.org. [retrieved: July 2014]. (Cited on pages 25, 26, and 52.)

Auto Scaling on the Google Cloud Platform (n.d.). https://cloud.google.com/developers/articles/auto-scaling-on-the-google-cloud-platform. [retrieved: March 2014]. (Cited on page 55.)

Bai, X., Xie, J., Chen, B. and Xiao, S. (2007). Dresr: Dynamic routing in enterprise service bus, *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pp. 528–531. (Cited on page 54.)

Bauer, D., Garces-Erice, L., Rooney, S. and Scotton, P. (2008). Toward scalable real-time messaging, *IBM Systems Journal* **47**(2): 237–250.

Benosman, R., Albrieux, Y. and Barkaoui, K. (2012). Performance evaluation of a massively parallel esb-oriented architecture, *Service-Oriented Computing and Applications (SOCA), 2012 5th IEEE International Conference on*, pp. 1–4.

Calinescu, R., Grunske, L., Kwiatkowska, M., Mirandola, R. and Tamburrelli, G. (2011). Dynamic qos management and optimization in service-based systems, *Software Engineering, IEEE Transactions on* **37**(3): 387–409. (Cited on page 54.)

Chang, S.-H., La, H. J., Bae, J. S., Jeon, W. Y. and Kim, S. D. (2007). Design of a dynamic composition handler for esb-based services, *e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on*, pp. 287–294. (Cited on page 54.)

Chappell, D. (2004). *Enterprise Service Bus*, O'Reilly Media, Inc., Sebastopol, CA, USA. (Cited on pages vi, 11, 12, 13, and 54.)

Chen, S. and Greenfield, P. (2004). QoS Evaluation of JMS: An Empirical Approach, *HICSS '04: Proceedings of the Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 9*, IEEE Computer Society, Washington, DC, USA, p. 90276.2.

Conrad, S., Hasselbring, W., Koschel, A. and Tritsch, R. (2006). *Enterprise Application Integration: Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele*, Elsevier, Spektrum, Akad. Verl. (Cited on pages 7 and 49.)

Devaram, K. and Andresen, D. (2003). SOAP optimization via parameterized client-side caching, *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, pp. 785–790.

Didona, D., Carnevale, D., Galeani, S. and Romano, P. (2012). An extremum seeking algorithm for message batching in total order protocols, *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference on*, pp. 89–98. (Cited on page 54.)

Engels, G., Hess, A., Humm, B., Juwig, O., Lohmann, M., Richter, J.-P., Voß, M. and Willkomm, J. (2008). *Quasar Enterprise - Anwendungslandschaften serviceorientiert gestalten*, dpunkt Verlag. (Cited on page 14.)

Estrella, J. C., Santana, M. J., Santana, R. H. C. and Monaco, F. J. (2008). Real-Time Compression of SOAP Messages in a SOA Environment, *SIGDOC '08: Proceedings of the 26th annual ACM international conference on Design of communication*, ACM, New York, NY, USA, pp. 163–168.

EXI Working Group [online]. 2007. Available from: `http://www.w3.org/XML/EXI` [cited January 2008]. (Cited on page 15.)

Fleck, J. (1999). A distributed near real-time billing environment, *Telecommunications Information Networking Architecture Conference Proceedings, 1999. TINA '99*, pp. 142–148. (Cited on pages 3 and 47.)

Friedman, R. and Hadad, E. (2006). Adaptive batching for replicated servers, *Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on*, pp. 311–320. (Cited on page 54.)

Friedman, R. and Renesse, R. V. (1997). Packing messages as a tool for boosting the performance of total ordering protocls, *Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing*, HPDC '97, IEEE Computer Society, Washington, DC, USA, pp. 233–. (Cited on page 54.)

Garces-Erice, L. (2009). Building an enterprise service bus for real-time soa: A messaging middleware stack, *Computer Software and*

Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International, Vol. 2, pp. 79–84.

González, L. and Ruggia, R. (2011). Addressing qos issues in service based systems through an adaptive esb infrastructure, *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*, MW4SOC '11, ACM, New York, NY, USA, pp. 4:1–4:7. Available from: http://doi.acm.org/10.1145/2093185.2093189. (Cited on page 54.)

Habich, D., Richly, S. and Grasselt, M. (2007). Data-Grey-Box Web Services in Data-Centric Environments, *IEEE International Conference on Web Services, 2007. ICWS 2007*, pp. 976–983.

Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W. and Maier, A. (2007). BPEL-DT – Data-Aware Extension of BPEL to Support Data-Intensive Service Applications, *Emerging Web Services Technology* **2**: 111–128.

Haesen, R., Snoeck, M., Lemahieu, W. and Poelmans, S. (2008). On the definition of service granularity and its architectural impact, *in* Z. Bellahsène and M. Léonard (eds), *Advanced Information Systems Engineering*, Vol. 5074 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pp. 375–389. Available from: http://dx.doi.org/10.1007/978-3-540-69534-9_29.

Henjes, R., Menth, M. and Zepfel, C. (2006). Throughput Performance of Java Messaging Services Using WebsphereMQ, *ICDCSW '06: Proceedings of the 26th IEEE International ConferenceWorkshops on Distributed Computing Systems*, IEEE Computer Society, Washington, DC, USA, p. 26.

Her, J. S., Choi, S. W., Oh, S. H. and Kim, S. D. (2007). A Framework for Measuring Performance in Service-Oriented Architecture, *NWESP '07: Proceedings of the Third International Conference on Next Generation Web Services Practices*, IEEE Computer Society, Washington, DC, USA, pp. 55–60.

Hess, A., Humm, B. and Voß, M. (2006). Regeln für serviceorientierte Architekturen hoher Qualität, *Informatik Spektrum* **29**(6): 395–411. (Cited on pages 15 and 16.)

Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. (Cited on pages vi, 7, 25, 50, and 51.)

IEEE (2008). IEEE standard for a precision clock synchronization protocol for networked measurement and control systems, *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* pp. c1–269. (Cited on page 30.)

Jongtaveesataporn, A. and Takada, S. (2010). Enhancing enterprise service bus capability for load balancing, *W. Trans. on Comp.* **9**(3): 299–308. Available from: http://dl.acm.org/citation.cfm?id=1852392.1852401. (Cited on page 54.)

Josuttis, N. (2007). *SOA in practice*, O'Reilly, Sebastopol, CA, USA. (Cited on pages 14 and 15.)

Krafzig, D., Banke, K. and Slama, D. (2005). *Enterprise SOA*, Prentice Hall. (Cited on page 14.)

Menth, M., Henjes, R., Zepfel, C. and Gehrsitz, S. (2006). Throughput Performance of Popular JMS Servers, *SIGMETRICS '06/Performance '06: Proceedings of the joint international conference on Measurement and modeling of computer systems*, ACM, New York, NY, USA, pp. 367–368.

Ng, A. (2006). Optimising Web Services Performance with Table Driven XML, *ASWEC '06: Proceedings of the Australian Software Engineering Conference*, IEEE Computer Society, Washington, DC, USA, pp. 100–112.

Ng, A., Greenfield, P. and Chen, S. (2005). A Study of the Impact of Compression and Binary Encoding on SOAP Performance, *Proceedings of the Sixth Australasian Workshop on Software and System Architectures (AWSA2005)*.

O'Brien, L., Merson, P. and Bass, L. (2007). Quality Attributes for Service-Oriented Architectures, *SDSOA '07: Proceedings of the International Workshop on Systems Development in SOA Environments*, IEEE Computer Society, Washington, DC, USA, p. 3. (Cited on page 13.)

PTP daemon (PTPd) (2013). http://ptpd.sourceforge.net. [retrieved: July 2014]. (Cited on page 30.)

Richter, J.-P., Haller, H. and Schrey, P. (2005). Serviceorientierte Architektur, *Informatik Spektrum* **28**(5): 413–416. (Cited on page 11.)

Romano, P. and Leonetti, M. (2012). Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning, *Computing, Networking and Communications (ICNC), 2012 International Conference on*, pp. 786–792. (Cited on page 54.)

Sachs, K., Kounev, S., Bacon, J. and Buchmann, A. (2009). Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark, *Perform. Eval.* **66**(8): 410–434.

Sachs, K., Kounev, S., Carter, M. and Buchmann, A. (2007). Designing a Workload Scenario for Benchmarking Message-Oriented Middleware, *Proceedings of the 2007 SPEC Benchmark Workshop, SPEC*.

Schulte, R. (2002). Predicts 2003: Enterprise Service Buses Emerge, Gartner. (Cited on page 11.)

SOAP Specification [online]. 2007. Available from: http://www.w3.org/TR/soap [cited January 2008]. (Cited on page 13.)

Spring Batch (2013). http://static.springsource.org/spring-batch/. [retrieved: July 2014]. (Cited on page 24.)

Suzumura, T., Takase, T. and Tatsubori, M. (2005). Optimizing Web Services Performance by Differential Deserialization, *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, IEEE Computer Society, Washington, DC, USA, pp. 185–192.

Tekli, J., Damiani, E., Chbeir, R. and Gianini, G. (2012). Soap processing performance and enhancement, *Services Computing, IEEE Transactions on* **5**(3): 387–403.

Ueno, K. and Tatsubori, M. (2006). Early capacity testing of an enterprise service bus, *Web Services, 2006. ICWS '06. International Conference on*, pp. 709–716.

Wichaiwong, T. and Jaruskulchai, C. (2007). A Simple Approach to Optimize Web Services' Performance, *NWESP '07: Proceedings of the Third International Conference on Next Generation Web Services Practices*, IEEE Computer Society, Washington, DC, USA, pp. 43–48.

Wu, B., Liu, S. and Wu, L. (2008). Dynamic reliable service routing in enterprise service bus, *Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE*, pp. 349–354. (Cited on page 54.)

Zhuang, Z. and Chen, Y.-M. (2012). Optimizing jms performance for cloud-based application servers, *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pp. 828–835.

Ziyaeva, G., Choi, E. and Min, D. (2008). Content-based intelligent routing and message processing in enterprise service bus, *Convergence and Hybrid Information Technology, 2008. ICHIT '08. International Conference on*, pp. 245–249. (Cited on page 54.)

PUBLICATIONS

Swientek, M., Bleimann, U. and Dowland, P. (2008). Service-Oriented Architecture: Performance Issues and Approaches, *in* P. Dowland and S. Furnell (eds), *Proceedings of the Seventh International Network Conference (INC2008)*, University of Plymouth, Plymouth, UK, pp. 261–269.

Swientek, M., Humm, B., Bleimann, U. and Dowland, P. (2009). An SOA Middleware for High-Performance Communication, *in* U. Bleimann, P. Dowland, S. Furnell and V. Grout (eds), *Proceedings of the Fifth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2009)*, University of Plymouth, Plymouth, UK.

Swientek, M., Humm, B., Bleimann, U. and Dowland, P. (2014). An Adaptive Middleware for Near-Time Processing of Bulk Data, *ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications*, Venice, Italy, p. 37 to 41.