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 systems**?**. 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

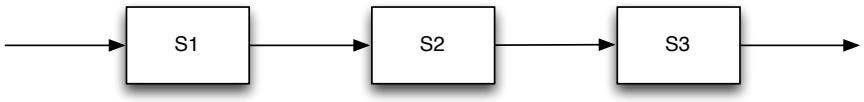
1.2 AIMS AND OBJECTIVES OF THE RESEARCH

1.3 CONTRIBUTIONS

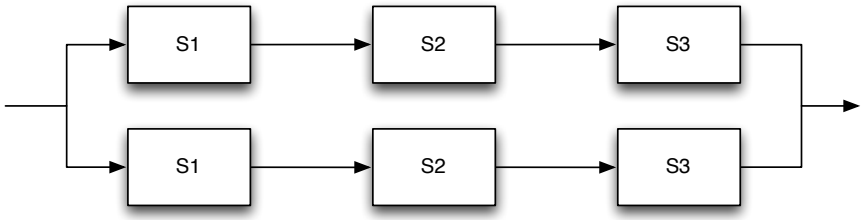
1.4 OUTLINE OF THE THESIS

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 1a).



(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 2). 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:

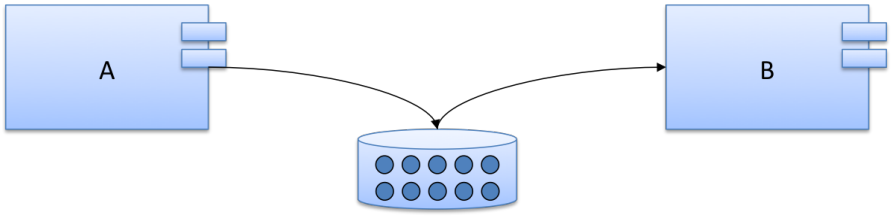


Figure 2: Batch processing

- **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 3). A messaging server or message-oriented middleware handles the asynchronous ex-

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

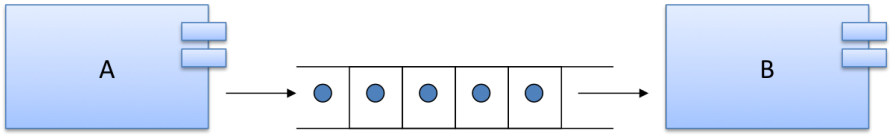


Figure 3: Message-based processing

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.

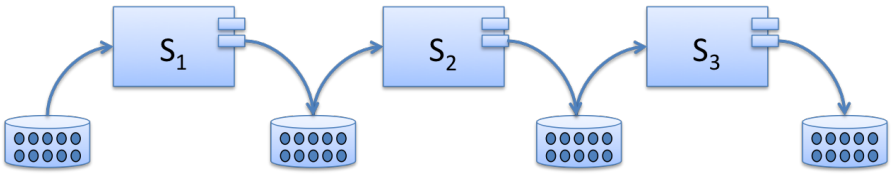


Figure 4: Batch processing system comprised of three subsystems

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

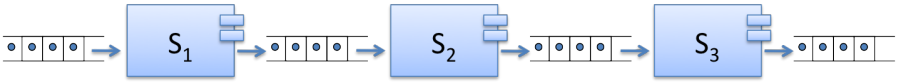


Figure 5: Message-based system comprised of three subsystems

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.

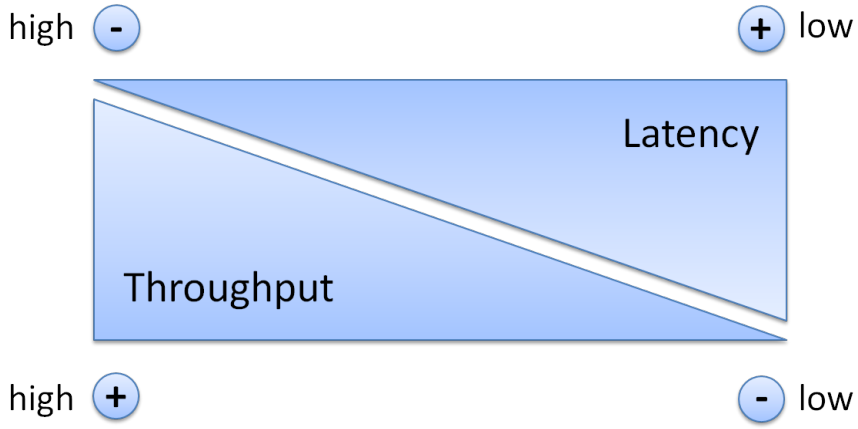


Figure 6: Latency and throughput are opposed to each other

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.

Part II

CONTRIBUTIONS

3

PERFORMANCE EVALUATION OF BATCH AND MESSAGE-BASED SYSTEMS

3.1 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.

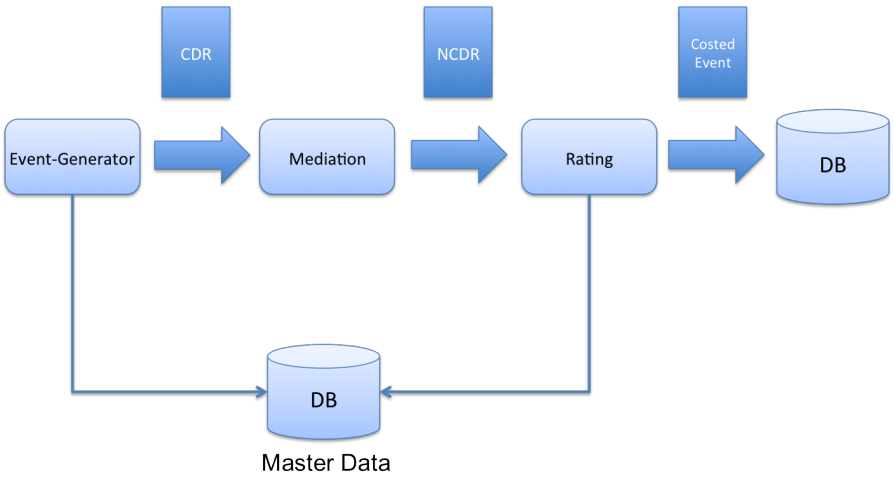


Figure 8: Components of the billing application prototype

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.

3.1.1 *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).

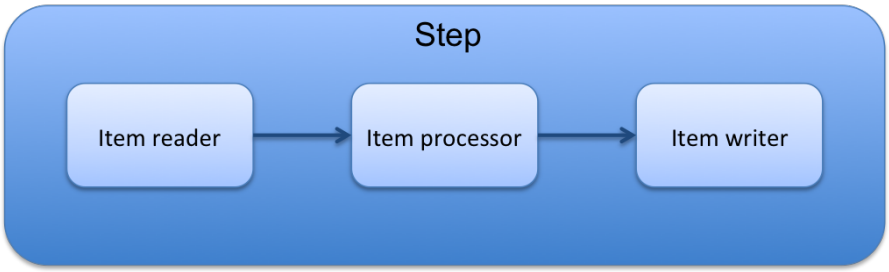


Figure 9: A Step consists of an item reader, item processor and item writer

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*.

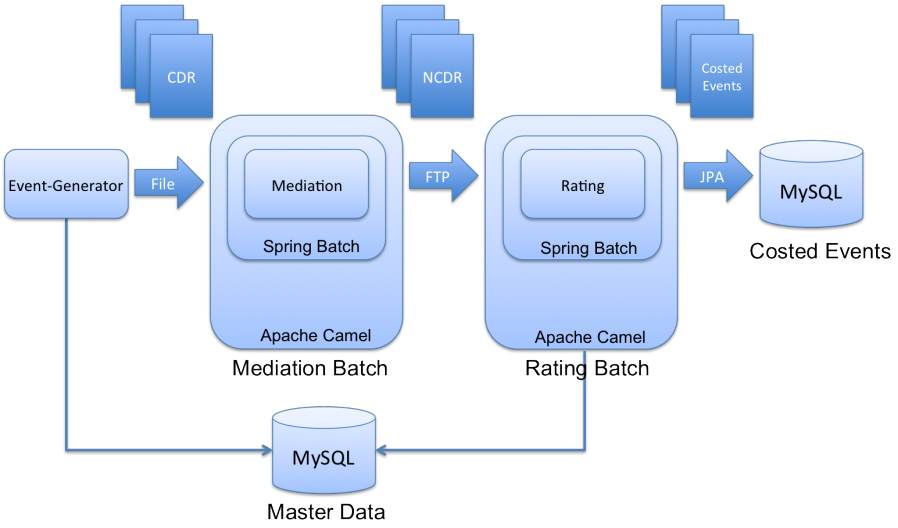


Figure 10: Batch prototype

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

3.1.2 *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 11 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.

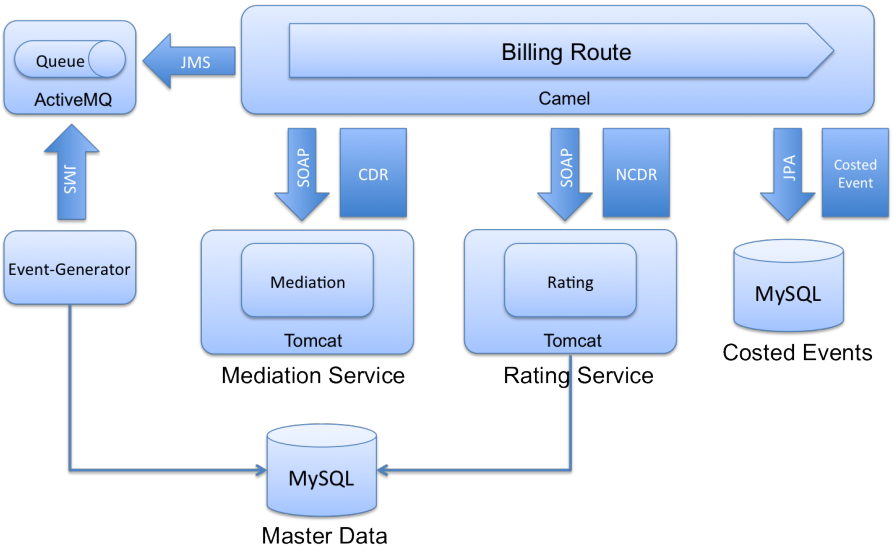


Figure 11: Message-based prototype

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.

3.2 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.

3.2.1 *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.

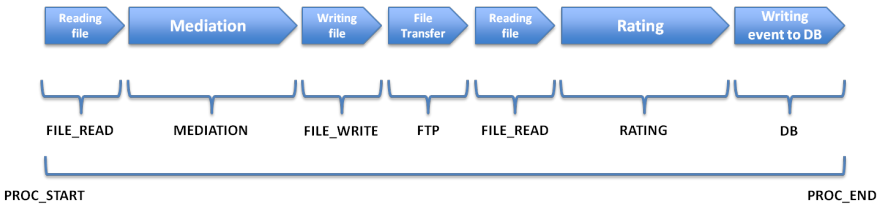


Figure 12: Measuring points of the batch prototye

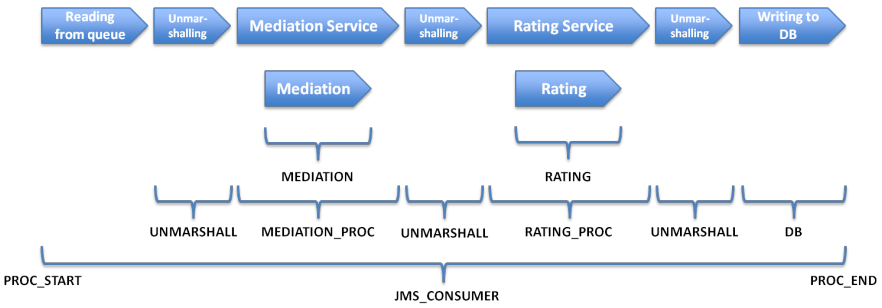


Figure 13: Measuring points of the messaging prototype

Table 1: 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 1 and 2.

3.2.2 *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 2: 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.

3.2.3 *Test environment*

The two prototypes have been deployed to an Amazon EC2 environment to conduct the performance evaluation, with the characterics described in Table 3.

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.

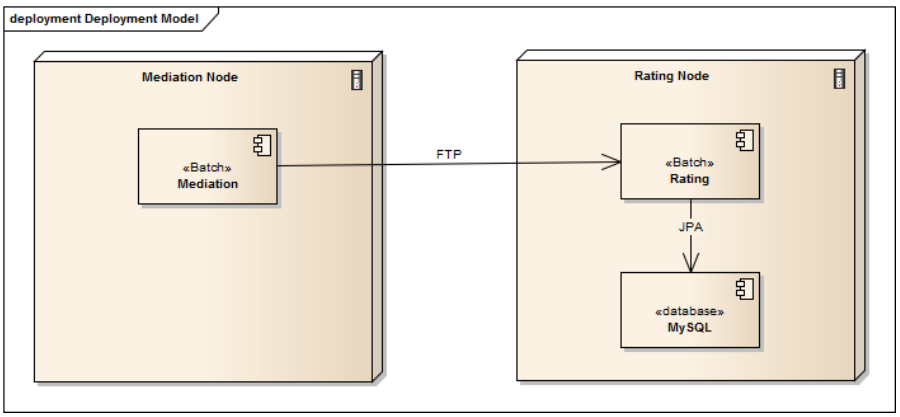


Figure 14: Batch prototype deployment on EC2 instances

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.

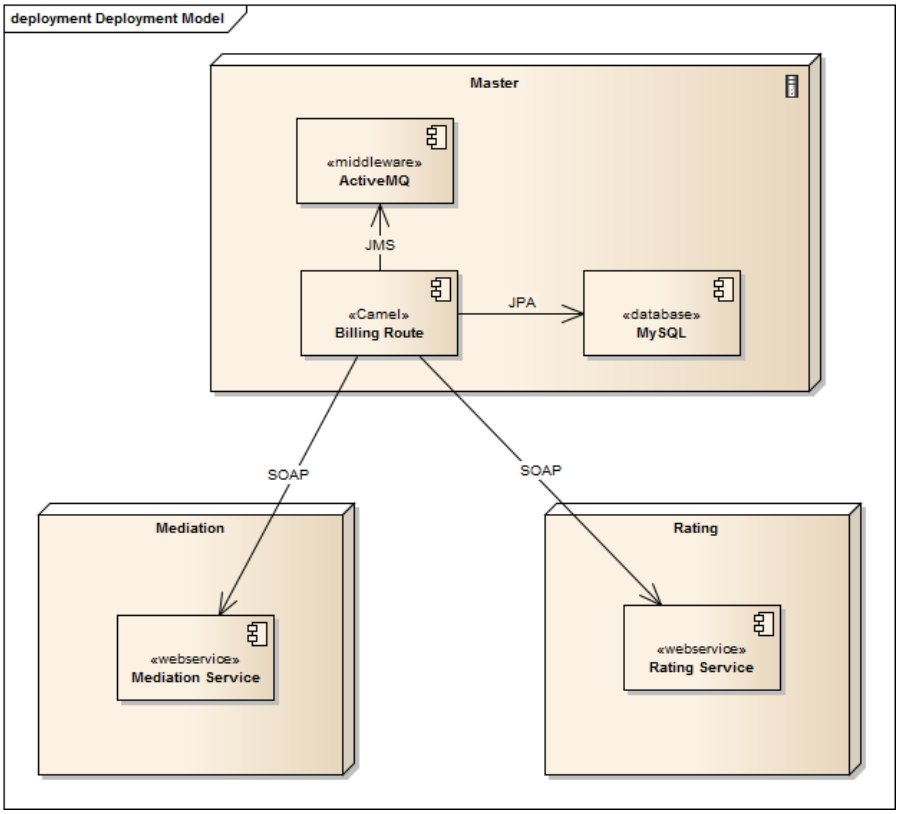


Figure 15: Messaging prototype deployment on EC2 instances

3.2.4 *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 3: 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.

3.2.5 *Results*

The performance evaluation yields the following results.

3.2.5.1 *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.

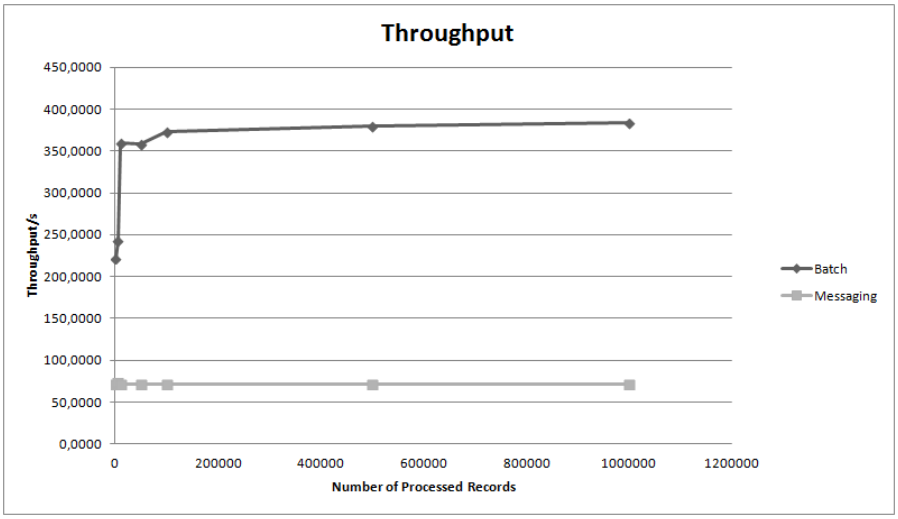


Figure 16: Throughput

3.2.5.2 *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.

3.2.5.3 *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

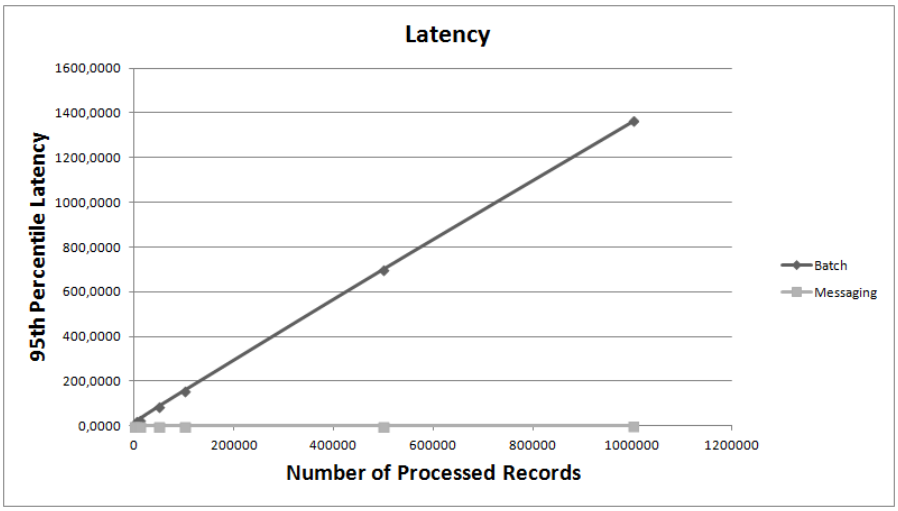


Figure 17: Latency

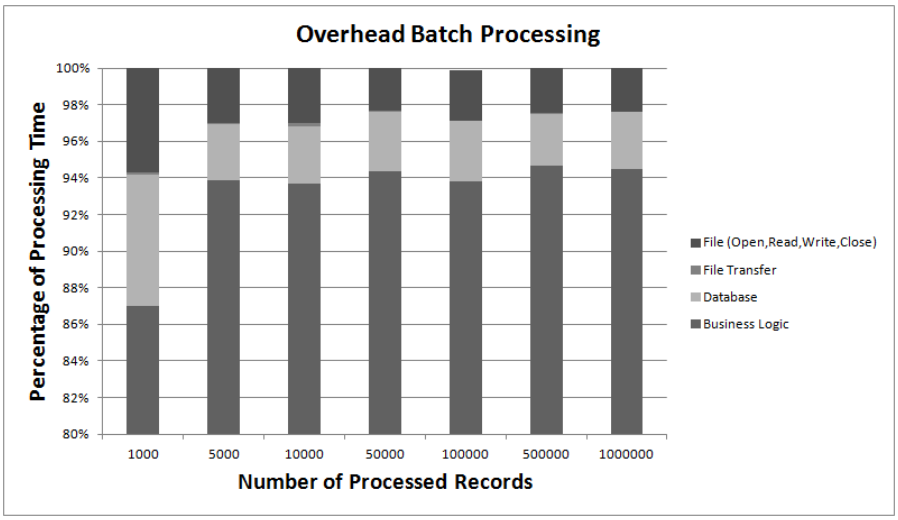


Figure 18: Overhead batch prototype

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.

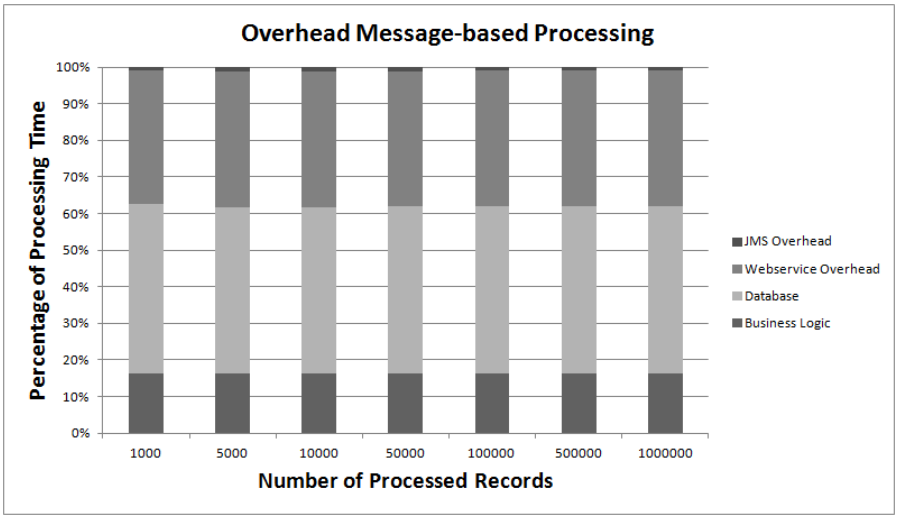


Figure 19: Overhead messaging prototype

3.2.5.4 *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

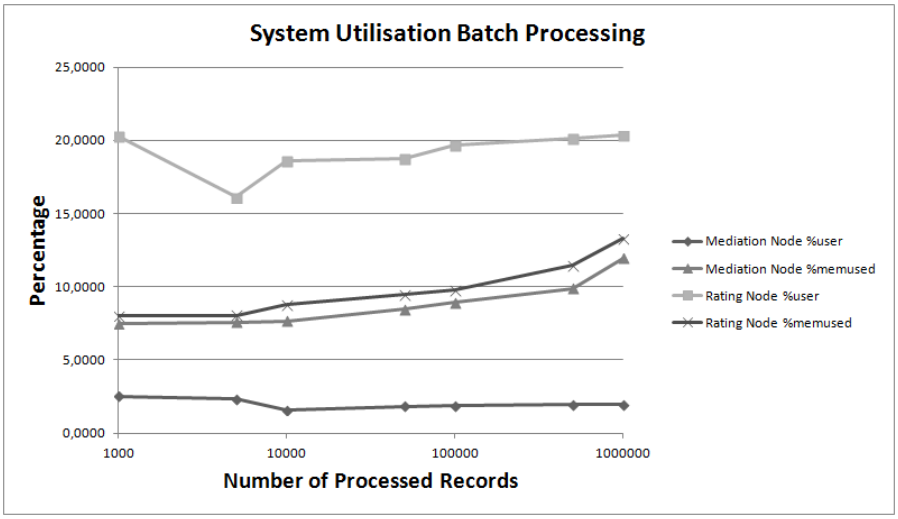


Figure 20: System utilisation batch prototype

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.

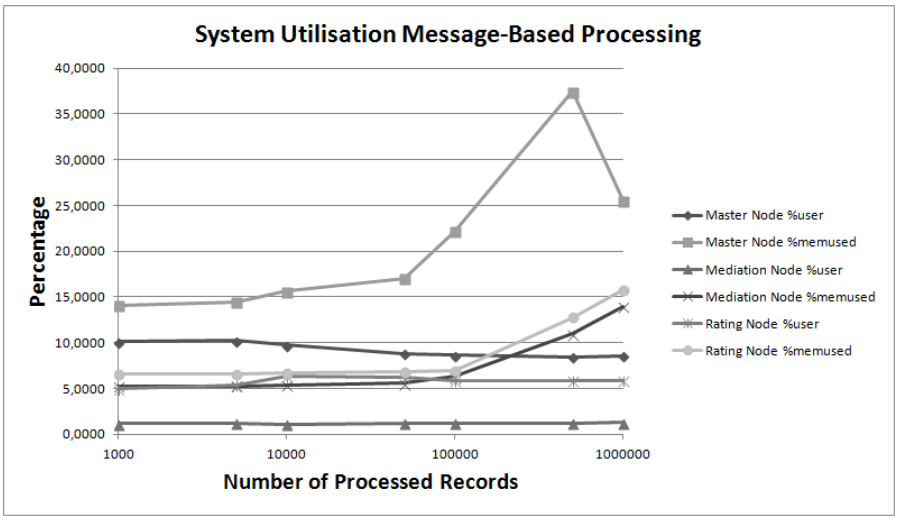


Figure 21: System utilisation messaging prototype

3.3 IMPACT OF DATA GRANULARITY ON THROUGHPUT AND LATENCY

The results presented in Section 3.2.5 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.

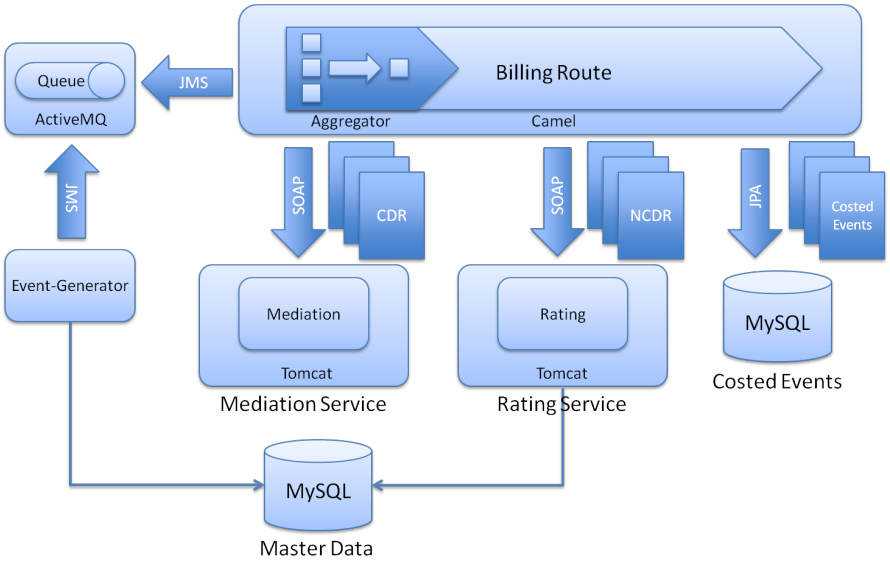


Figure 22: The data granularity is controlled by an aggregator

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 23 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 $aggregation\_size > 50$.

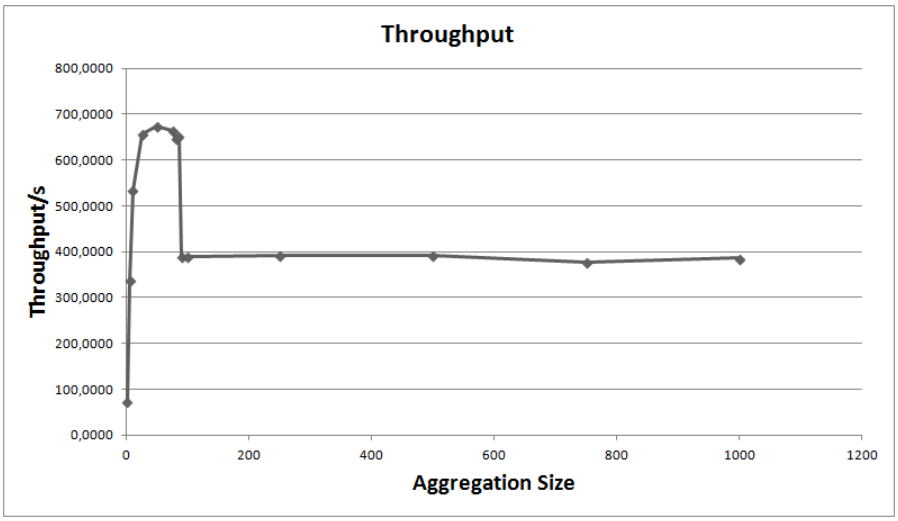


Figure 23: Impact of different aggregation sizes on throughput

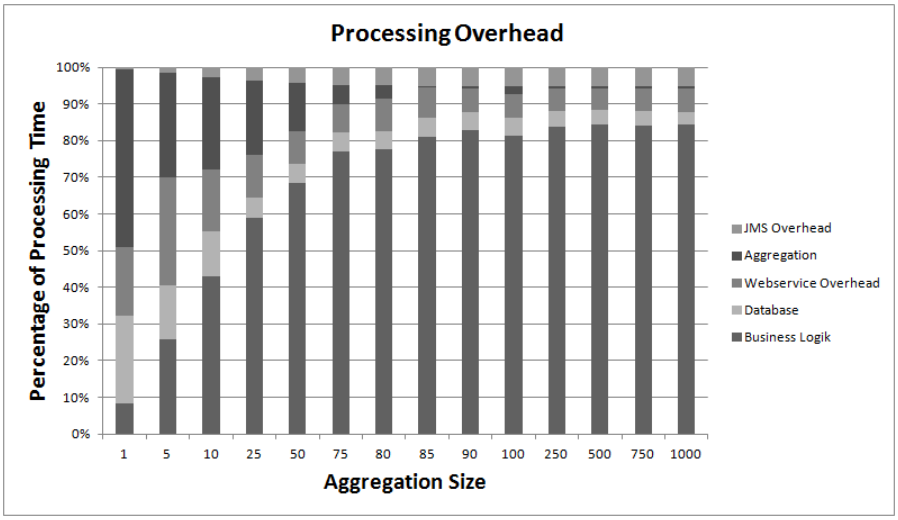


Figure 24: Impact of different aggregation sizes on processing overhead

The increased throughput achieved by increasing the aggregation size comes with the cost of a higher latency. Figure 25 shows the impact of different aggregation sizes on the 95th percentile latency of the messaging prototype.

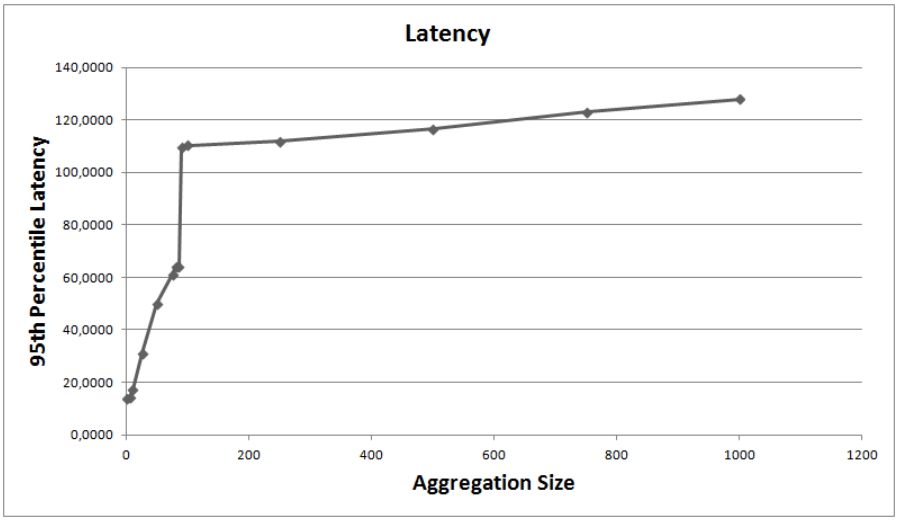


Figure 25: Impact of different aggregation sizes on latency

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 3.2.5.2).

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.

3.4 SUMMARY

Near-time processing of bulk data is hard to achieve. As shown in Section 2.3, latency and throughput are opposed performance metrics of a system for bulk data processing. High throughput, as provided

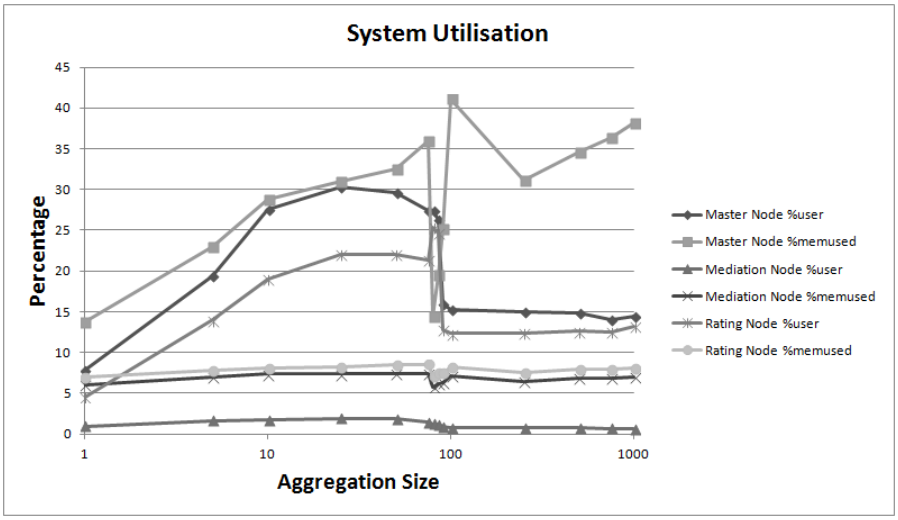


Figure 26: Impact of different aggregation sizes on system utilisation

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 3.3 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.

4

AN ADAPTIVE MIDDLEWARE FOR NEAR-TIME PROCESSING OF BULK DATA

5

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

Part III

CONCLUSION

BIBLIOGRAPHY

Apache Camel (2014). http://camel.apache.org. [retrieved: July 2014]. (Cited on pages 15 and 16.)

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

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 7 and 15.)

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 20.)

PTP daemon (PTPd) (2013). http://ptpd.sourceforge.net. [retrieved: July 2014]. (Cited on page 20.)

Spring Batch (2013). http://static.springsource.org/spring-batch/. [retrieved: July 2014]. (Cited on page 14.)

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.