A Feedback-Controlled Adaptive Middleware for Near-Time Bulk Data Processing

Martin Swientek
Paul Dowland

School of Computing and Mathematics
Plymouth University
Plymouth, UK
e-mail: {martin.swientek, p.dowland}@plymouth.ac.uk

Bernhard Humm
Udo Bleimann

Department of Computer Science
University of Applied Sciences Darmstadt
Darmstadt, Germany
e-mail: {bernhard.humm, udo.bleimann}@h-da.de

***Abstract*—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 over time. In this article, the concept of a middleware is introduced that adapts its processing type fluently between batch processing and single-event processing using a feedback-control loop. By adjusting the data granularity at runtime, the system is able to minimize the end-to-end latency for different load scenarios. The proposed middleware concept has been implemented with a research prototype and has been evaluated. The results of the evaluation show that the concept is viable and is able to optimize the end-to-end latency of a system for bulk data processing.**

Keywords–adaptive middleware; message aggregation; latency; throughput.

I. Introduction

This article extends previous work in [1]. 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 [2]. 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 a 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. An Example: Billing Systems for Telecommunications Carriers

An example of a system for bulk data processing is a billing system of a telecommunications carrier. 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 1).

The performance requirements of such a billing system are high. It has to process more than 1 million records per hour and the whole batch run needs to be finished in a limited timeframe to comply with service level agreements with the print service provider. Since delayed invoicing causes direct loss of cash, it has to be ensured that the bill arrives at the customer on time.

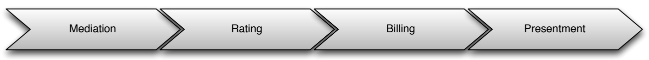


Figure 1. Billing process

B. Near-Time Processing of Bulk Data

A new requirement for systems for bulk data processing is near-time processing. Near-time processing aims to reduce the end-to-end latency of a business process, that is, the time that is spent between the occurrence of an event and the end of its processing. In case of a billing system, it is the time between the user making a call and the complete processing of this call including mediation, rating, billing and presentment.

The need for near-time charging and billing for telecommunications carriers is induced by market forces, such as the increased advent of mobile data usage and real-time data services [3]. Carriers want to offer new products and services that require real-time or near-time charging and billing. Customers want more transparency, for example, to set their own limits and alerts for their data usage, which is currently only possible for pre-paid accounts. Currently, a common approach for carriers is to operate different platforms for real-time billing of pre-paid accounts and traditional batch-oriented billing for post-paid accounts. To reduce costs, carriers aim to converge these different platforms.

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. A system is therefore either optimized for low latency or high maximum throughput. 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 article, a solution to this problem is proposed:

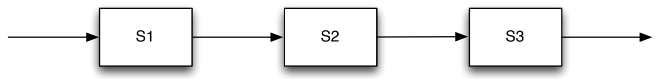
- The concept of a middleware is presented that is able to adapt its processing type fluently between batch processing and single-event processing. By adjusting the data granularity at runtime, the system is able to minimize the end-to-end latency for different load scenarios.
- A prototype has been built to evaluate the concepts of the adaptive middleware.
- A performance evaluation has been conducted using this prototype to evaluate the proposed concept of the adaptive middleware.

This article extends the adaptive middleware concept, which has been presented in [1]. It adds a discussion of its underlying concepts and design aspects, that should be considered when implementing such an adaptive middleware for near-time processing of bulk data. In addition, it describes the prototype implementation of the middleware concept and presents the results of the evalution of the propposed approach, as well as its limitations.

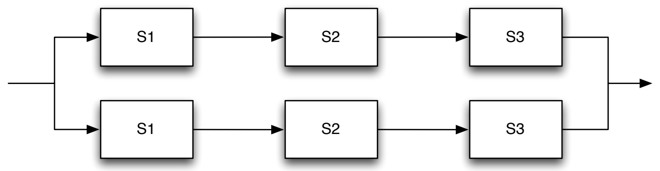
The remainder of this article is organized as follows. Section II defines the considered type of system and the terms throughput and latency. Section III gives an overview of other work related to this research. The concept, components and design aspects of the adaptive middleware are presented in Section IV through VI. Section VII describes the prototype system that has been build to evaluate the proposed concepts. The evaluation of the prototype system is presented in Section VIII. Section IX describes the limitations of this research. Finally, Section X concludes the paper and gives and outlook to further research.

II. 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 2a).



(a) Single processing line



(b) Parallel processing lines

Figure 2. 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 being distributed among each line. For simplification, a system with a single processing line is considered in the remainder of this article.

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

A. Batch processing

The traditional operation paradigm of a system for bulk data processing is batch processing (see Figure 3). 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.

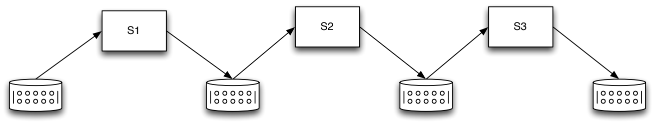


Figure 3. Batch processing

B. Message-base processing

Messaging facilitates the integration of heterogeneous applications using asynchronous communication. Applications are communicating with each other by sending messages (see Figure 4). A messaging server or message-oriented middleware handles the asynchronous exchange of messages including an appropriate transaction control [4].

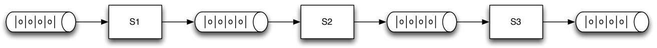


Figure 4. Message-based processing

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.

C. 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 article:

- **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 article 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.

III. Related Work

This section gives an overview of work related to the research presented in this article. It discusses performance optimizations in the context of transport optimization, middleware optimizations and message batching.

The proposed middleware for high-performance near-time processing of bulk data adjusts the data granularity itself at runtime. Work on middleware discusses different approaches for self-adjustment and self-awareness of middleware, which can be classified as adaptive or reflective middleware.

Automatic scaling of server instances is another approach to handle infrequent load spikes. Additionally, the section gives a brief overview of feedback-control of computing systems.

Research on messaging middleware currently focusses on Enterprise Service Bus (ESB) infrastructure. An ESB is an integration platform that combines messaging, web services, data transformation and intelligent routing to connect multiple heterogeneous services [5]. 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.

A. Transport Optimization

Most of the work that aims to optimize the performance of service-oriented systems is done in the area of Web Services since it is a common technology to implement a SOA.

In particular, various approaches have been proposed to optimize the performance of SOAP, the standard protocol for Web Service communication. This includes approaches for optimizing the processing of SOAP messages (cf. [6] [7] [8]), compression of SOAP messages (cf. [9] [10]) and caching (cf. [11] [12]). A survey of the current approaches to improve the performance of SOAP can be found in [13].

[14] proposes an approach to transfer bulk data between web services per File Transfer Protocol (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 [15]. Instead of transferring the data wrapped in SOAP messages, it is transferred using an external data layer. For example, when using database systems as a 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, [16] developed BPEL data transitions to explicitly specify data flows in BPEL processes.

[17] proposes three tuning strategies to improve the performance of Java Messaging Service (JMS) for cloud-based applications.

1) When using persistent mode for reliable messaging the storage block size should be matched with the message size to maximize 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

In contrast, the optimization approach presented in this thesis is aimed at the integration layer of messaging system, which allows further optimizations, such as dynamic message batching and message routing.

B. Middleware Optimizations

Some research has been done to add real-time capabilities to ESB or messaging middleware. [18] proposes an architecture for a real-time messaging middleware based on an ESB. 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 survey [19], the authors describe a real-time ESB model by extending the Java Business Integration (JBI) specification with semantics for priority and time restrictions and modules for flow control and bandwidth allocation. The proposed system is able to dynamically allocate bandwidth according to business requirements.

MPAB (Massively Parallel Application Bus) is an ESB-oriented messaging bus used for the integration of business applications [20]. 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.

Tempo is a real-time messaging system written in Java that can be used on either a real-time or non-real-time architecture [21]. 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 throughput. The main principle of Tempo is to use an independent queuing system for each topic. Resources 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.

In contrast to these approaches, the approach presented in this thesis is based on a standard middleware and can be used with several integration technologies, such as JMS or SOAP.

C. Message Batching

Aggregating or batching of messages is a common approach for optimizing performance and has been applied to several domains. TCP Nagle's algorithm is a well-known example of this approach [22].

Message batching for optimizing the throughput of Total Ordering Protocols (TOP) have first been investigated by [23]. In their work, the authors have compared the throughput and latency of four different Total Ordering Protocols. They conclude that "batching messages is the most important optimization a protocol can offer".

[24] extends the work of [23] with a policy for varying the batch level automatically, based on dynamic estimates of the optimal batch level.

[25] presents a mechanism for self-tuning the batching level of Sequencer-based Total Order Broadcast Protocols (STOB), that combines analytical modeling an Reinforcement Learning (RL) techniques.

[26] proposes a self-tuning algorithm based on extremum seeking optimization principles for controlling the batching level of a Total Order Broadcast algorithm. It uses multiple instances of extremum seeking optimizers, each instance is associated with a distinct value of batching b and learns the optimal waiting time for a batch of size b.

[27] describes two generic adaptive batching schemes for replicated servers, which adapt their batching level automatically and immediately according to the current communication load, without any explicit monitoring of the system.

The approach presented in this research applies the concept of dynamic message batching to minimize the end-to-end latency of a message-based system for bulk data processing.

D. Self-Adaptive Middleware

[28] argues that "the most adequate level and natural locus for applying adaption is at the middleware level". Adaption at the operating system level is platform-dependent and changes at this level affect every application running on the same node. On the other hand, adaption at application level assigns the responsibility to the developer and is also not reusable.

[29] proposes an adaptive, general-purpose runtime infrastructure for effective resource management of the infrastructure. Their approach is comprised of three components:

1) dynamic performance prediction
2) adaptive intra-site performance management
3) adaptive inter-site resource management

The runtime infrastructure is able to choose from a set of performance predictions for a given service and to dynamically choose the most appropriate prediction over time by using the prediction history of the service.

AutoGlobe [30] provides a platform for adaptive resource management comprised of

1) Static resource management
2) Dynamic resource management
3) Adaptive control of Service Level Agreements (SLA)

Static resource management optimizes the allocation of services to computing resources and is based on on automatically detected service utilisation patterns. Dynamic resource management uses a fuzzy controller to handle exceptional situations at runtime. The Adaptive control of Service Level Agreements (SLAs) schedules service requests depending on their SLA agreement.

The coBRA framework proposed by [31] is an approach to replace service implementations at runtime as a foundation for self-adaptive applications. The framework facilitates the replacement of software components to switch the implementation of a service with the interface of the service staying the same.

DREAM (Dynamic Reflective Asynchronous Middleware) [32] is a component-based framework for the construction of reflective Message-Oriented Middleware. Reflective middleware "refers to the use of a causally connected self-presentation to support the inspection and adaption of the middleware system" [33]. DREAM is based on FRACTAL, a generic component framework and supports various asynchronous communication paradigms such as message passing, event-reaction and publish/subscribe. It facilitates the construction and configuration of Message-Oriented Middleware from a library of components such as message queues, filters, routers and aggregators, which can be assembled either at deploy-time or runtime.

E. Adaption in Service-Oriented Architectures

Several adaption methods have been proposed in the context of service-based applications. In their survey [34], the authors describe the following adaption methods:

- **Adaption by Dynamic Service Binding**
  This adaption method relies on the ability to select and dynamically substitute services at run-time or at deployment-time. Services are selected in such a way that the adaption requirements are satisfied in the best possible way.
- **Quality of Service (QoS)-Driven Adaption of Service Compositions**
  The goal of this adaption approach is to select the best set of services available at run-time, under consideration of process constraints, end-user preferences and the execution context.
- **Adaption of Service Interfaces and Protocols**
  The goal of this adaption approach is to mediate between two services with different signatures, interfaces and protocols. This includes signature-based adaption, ontology-based adaption or behavior-based adaption.

F. Adaptive ESB

Research on messaging middleware currently focusses on ESB infrastructure. An ESB is an integration platform that combines messaging, web services, data transformation and intelligent routing to connect multiple heterogeneous services [5]. 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 work 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 [35], routing [36] [37] [38] and load balancing [39].

The DRESR (Dynamic Reconfigurable ESB Service Routing), proposed by [36], allows the routing table to be changed dynamically at run-time based on service selection preferences, such as response time. It defines mechanisms to test and evaluate the availability and performance of a service and to select services based on its testing results and historical data.

[38] proposes a framework for content-based intelligent routing. It evaluates the service availability and selects services based on its content and properties.

[39] proposes a load balancing mechanism that distributes requests to services of the same *service type*, having the same function and signature, and enables the dynamic selection of the target service.

Work to manage and improve the 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 [40].

[41] proposes an adaptive ESB infrastructure to address 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.

In contrast to these solutions, the approach presented in this article uses dynamic message aggregation and message routing as adaption mechanism to optimize the end-to-end latency of messaging system for different load scenarios.

G. Automatic Scaling

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 [42] or Google App Engine [43]. While scaling is a common approach to improve the performance of a system, it also leads to additional operational and possible license costs. Additionally, it is often difficult to scale certain components or external dependencies of the system, such as databases or external services. Of course, the approach presented in this article can be combined with these auto-scaling approaches.

H. Feedback-control of Computing Systems

Feedback-control has been applied to several different domains of computing systems since the early 1990s, including data networks, operating systems, middleware, multimedia and power management (cf. [44]). Feedback-control of middleware systems include application servers, such as the Apache http-Server, database management systems, such as IBM Universal Database Server, and e-mail servers, such as the IBM Lotus Domino Server. [44] describes 3 basic control problems in this context:

- Enforcing service level agreements
- Regulate resource utilization
- Optimize the system configuration

Additionally, feedback-control has been applied recently to web environments, such as web servers and web services, application servers, including data flow control in J2EE servers, Repair Management in J2EE servers and improving the performance of J2EE servers and cloud environments (cf. [45]).

The *Adaptive Middleware* presented in this article utilizes a closed-feedback loop to control the aggregation size of the processed messages, depending on the current load of the system to minimize the end-to-end latency of the system. This is a novel approach that has not previously been investigated.

IV. Middleware Concepts

The adaptive middleware is based on the following core concepts: (1) message aggregation, (2) message routing, and (3) monitoring and control.

A. Message Aggregation

Message aggregation or batching of messages is the main feature of the adaptive middleware to provide a high maximum throughput. The aggregation of messages has the following goals:

- To decrease the overhead for each processed message
- To facilitate optimized processing

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.

In [1], a static aggregation size has been used to optimize the latency and the throughput of a system. This is not feasible for real systems, since the the latency and throughput also depends on the load of the system. Therefore, a dynamic aggregation size depending on the current load of the system is needed.

B. Message Routing

The goal of the message routing is to route the message aggregate to the appropriate service, which is either optimized for batch or single event processing, to allow for an optimized processing. Message routing depends on how messages are aggregated. Table I shows the different strategies of message routing.

TABLE I
STRATEGIES FOR MESSAGE ROUTING

Routing Strategy	Examples	Description
Technical routing	Aggregation size	Routing is based on the technical properties of a message aggregate.
Content-based routing	Customer segments (e.g. business customers or private customers)	Routing is based on the content of the message aggregate, that is, what type of messages are aggregated.

With high levels of message aggregation, it is not preferred to send the aggregated message payload itself over the message bus using Java Messaging 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.

Message routing can be static or dynamic:

- **Static routing:**
  Static routing uses static routing rules, that are not changed automatically.
- **Dynamic routing:**
  Dynamic routing adjusts the routing rules automatically at run-time, for example, depending on QoS properties of services. For example, see [36], [37] or [38].

C. Monitoring and Control

In order to optimize the end-to-end latency of the system, the middleware needs to constantly monitor the load of the system and control the aggregation size accordingly (see Figure 5).

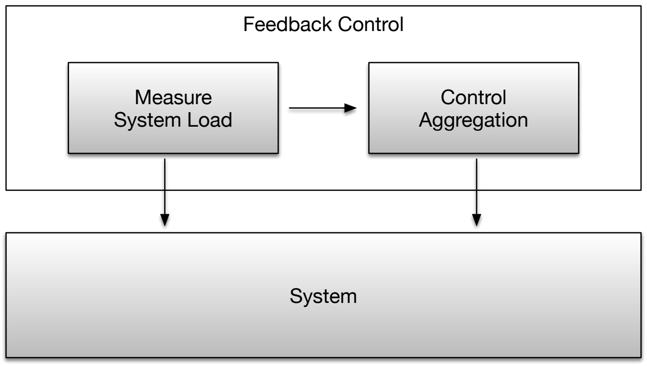


Figure 5. Monitoring and Control

If the current load of the system is low, the aggregation size should be small to provide a low end-to-end latency of the system. If the current load of the system is high, the aggregation size should be high to provide a high maximum throughput of the system.

To control the level of message aggregation at runtime, the adaptive middleware uses a closed feedback loop as shown in Figure 6, with the following properties:

- **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 for example, the average end-to-end latency.

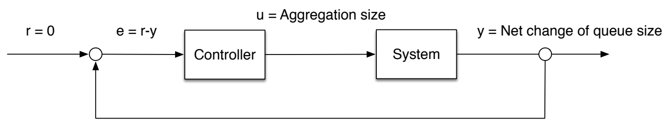


Figure 6. Feedback loop to control the aggregation size

V. MIDDLEWARE COMPONENTS

Figure 7 shows the components of the middleware, that are based on the Enterprise Integration Patterns described by [46]. A description of these components can be found in Table II.

VI. DESIGN ASPECTS

This section describes aspects that should be taken into account when designing an adaptive system for bulk data processing.

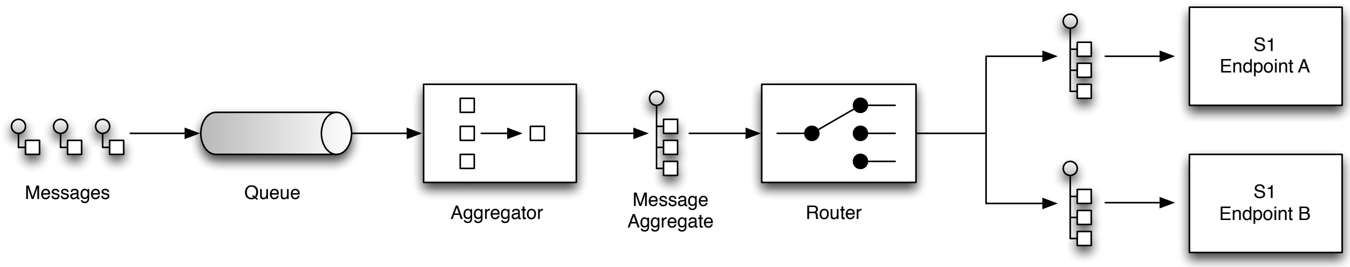
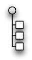
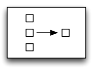
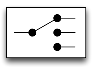
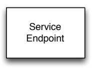


Figure 7. Middleware components

TABLE II
COMPONENTS OF THE ADAPTIVE MIDDLEWARE. WE ARE USING THE NOTATION DEFINED BY [46]

Symbol	Component	Description
	Message	A single message representing a business event.
	Message Aggregate	A set of messages aggregated by the Aggregator component.
	Queue	Storage component which stores messages using the FIFO principle.
	Aggregator	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.
	Router	Routes messages to the appropriate service endpoint.
	Service Endpoint	Represents a business service.

A. Service Design

The services that implement the business functionality of the system need to be explicitly designed to support the runtime adaption between single-event and batch processing.

There are different options for the design of these services:

- Single service interface with distinct operations for single and batch processing
  - The service provides different distinct operations for high and low aggregation sizes with optimized implementations for batch and single-event processing. The decision which operation should be called is done by the message router. It is generally not possible to use different transports for different aggregation sizes.
- Single service interface with a single operation for both single and batch processing
  - The service provides a single operation that is called for all aggregation sizes. The decision which optimization should be used is done by the service implementation. It is not possible to use different transports for different aggregation sizes.
- Multiple service interfaces for single and batch processing (or different aggregation sizes)
  - The logical business service is described by distinct service interfaces which contain operations for either batch processing or single-event processing. The decision which operation should be called is done by the message router. It is possible to use different transports for different aggregation sizes.

The choice of service design relates to where you want to have the logic for the message routing for optimized processing. With a single service offering distinct operations for single-event and batch processing, as well as with distinct service for each processing style, the message router decides which service endpoint should be called. In contrast, using a single service with a single operation for both processing styles, the service itself is responsible for choosing the appropriate processing strategy. Using a different integration type for each processing style is not possible in this case.

Listing 1 shows the interface of a service offering different operations for batch processing (line 6) and single-event processing (line 10).

B. Integration and Transports

The integration architecture defines the technologies that are used to integrate the business services. In general, different integration styles with different transports are used for batch processing and single-event processing, which needs to be taken into account when designing an adaptive system for bulk data processing.

When using high aggregation sizes, it is not feasible to use the same transports as with low aggregation sizes. Large messages should not be transferred over the messaging system. Instead, a file based transport using FTP or database-based integration should be used. When using a messaging system, the payload of large messages should not be transported over the messaging system. For example, by implementing the *Claim Check* Enterprise Integration Pattern (EIP) (cf. [46]).

Additionally, the technical data format should be considered.

The concrete threshold between low and high aggregation sizes depends on the integration architecture and implementation of the system, such as the integration architecture and the deployed messaging system.

Listing 1. Java interface of a web service offering different operations for single and batch processing.

```
@WebService
@SOAPBinding(style=Style.DOCUMENT, use=Use.LITERAL, parameterStyle=ParameterStyle.WRAPPED)
public interface RatingPortType {
  @WebMethod(operationName="processCallDetails")
  @WebResult(name="costedEvents")
  public Costedevents processCallDetails(@WebParam(name="callDetailRecords") SimpleCDRs
      callDetailRecords) throws ProcessingException, Exception;

  @WebMethod(operationName="processCallDetail")
  @WebResult(name="costedEvent")
  public Costedevent processCallDetail(@WebParam(name="simpleCDR") SimpleCDR callDetailRecord)
      throws ProcessingException, Exception;
}
```

TABLE III
TRANSPORT OPTIONS FOR HIGH AND LOW AGGREGATION SIZES

Aggregation Size	Transport Options
High	• Database • File-based (e.g. FTP) • Claim Check EIP
Low	• JMS • SOAP

The choice of the appropriate integration transport for a service is implicitly implemented by the message router (see Section IV-B).

C. Error Handling

Message aggregation has also an impact on the handling of errors that occur during the processing. Depending on the cause of the error, there are two common types of errors:

- **Technical errors**
  Technical errors are errors caused by technical reasons, for example, an external system is not available or does not respond within a certain timeout or the processed message has an invalid format.
- **Business errors**
  Business errors are caused by violation of business rules, for example, a call detail record contains a tariff that is no longer valid.

The following points should be taken into account, when designing the error handling for an adaptive system for bulk data processing:

- Write erroneous messages to an error queue for later processing.
- Use multiple queues for different types of errors, for example, distinct queues for technical and business errors to allow different strategies for handling them. Some type of errors can be fixed automatically, for example, an error that is caused by an outage of an external system, while other errors need to be fixed manually.
- If the erroneous messages is part of an aggregated message, it should be extracted from the aggregate to prevent the whole aggregate from beeing written to the error queue, especially when using high aggregation sizes.

D. Controller Design

There are several approaches for the implementation of feedback-control systems. [44] describes two major steps:

1) modeling the dynamics of the system
2) developing a control system

There are different approaches that are used in practice to model the dynamics of a system [47]:

- Empirical approach using curve fitting to create a model of the system
- Black-box modeling
- Modeling using stochastic approaches, especially queuing theory
- Modeling using special purpose representations, for example, the first principles analysis

For practical reasons, the following approach has been taken in this research:

1) Define the control problem
2) Define the input and output variables of the system
3) Measure the dynamics of the system
4) Develop the control system

1) Control Problem: The control problem is defined as follows:

- Minimize the end-to-end latency of the system by controlling the message aggregation size.
- The aggregation size used by the messaging system should depend on the current load of the system.
- When the system faces high load, the aggregation size should be increased to maximize the maximum throughput of the system.
- When the system faces low load, the aggregation size should be decreased to minimize the end-to-end latency of the system.

2) Input/Output Signals: [48] describes the following criteria for selecting input control signals:

- **Availability**
  It should be possible to influence the control input directly and immediately.

- **Responsiveness**
  The system should respond quickly to a change of the input signal. Inputs whose effect is subject to latency or delays should be avoided when possible.
- **Granularity**
  It should be possible to adjust the control input in small increments. If the control input can only be adjusted in fixed increments, then it could be necessary to consider this in the controller or actuator implementation.
- **Directionality**
  How does the control input impact the control output? Does an increased control input result in increased or decreased output?

Additionally, the following criteria should be considered for selecting output control signals:

- **Availability**
  The quantity must be observable without gaps and delays.
- **Relevance**
  The output signal should be relevant for the behavior of the system that should be controlled.
- **Responsiveness**
  The output signal should reflect changes of the state of the system quickly without lags and delays.
- **Smoothness**
  The output signal should be smooth and does not need to be filtered.

With regard to these criteria, the following input and output control signals have been chosen

- **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.

3) Control Strategy: We use a simple non-linear control strategy that could be implemented as follows (cf. [48]):

- When the tracking error is positive, increase the aggregation size by 1
- Do nothing when the tracking error is zero.
- Periodically decrease the aggregation size to test if a smaller queue size is able to handle the load.

VII. Prototype Implementation

To evaluate the proposed concepts of the adaptive middleware, a prototype of a billing system has been implemented using Apache Camel [49] as the messaging middleware.

Figure 8 shows the architecture of the 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 webservice 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.

TABLE IV
Technologies and frameworks used for the implementation of the prototypes

Language	Java 1.6
Dependency Injection	Spring 3.0.7
Persistence API	OpenJPA (JPA 2.0) 2.1.1
Database	MySQL 5.5.24
Logging	Logback 1.0.1
Test	JUnit 4.7
Batch Framework	Spring Batch 2.1.8
Messaging Middleware	Apache Camel 2.10.0
Other Frameworks	Joda-Time, Apache Commons

The prototypes are implemented with Java 1.6 using Java Persistence API (JPA) for the data-access layer and a MySQL database. See Table IV for complete list of technologies and frameworks used for the implementation of the prototypes.

The prototype performs the following steps:

1) The message is read from the billing queue using JMS. The queue is 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.
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 Database*.

The feedback-control loop of the prototype is implemented by the following components:

- **Performance Monitor**
  The *Performance Monitor* manages the feedback-control loop by periodically calling the *Sensor* and updating the *Controller*. Additionally, it calculates the current throughput and end-to-end latency of the system.
- **Sensor**
  The *Sensor* is responsible for getting the current size of the message queue using Java Management Extensions (JMX).
- **Controller**
  The *Controller* calculates the new value for the aggregation size based on the setpoint and the current error.
- **Actuator**
  The *Actuator* is responsible for setting the new aggregation size of the *Aggregator* calculated by the *Controller*.

A. Aggregator

The *Aggregator* is configured to dynamically use the aggregation size (*completionSize*) set by a message header, as shown in Listing 2 (line 2). This message header is set by the *Actuator* (see Section VII-B3), which is controlled by the *Controller* (see Section VII-B2).

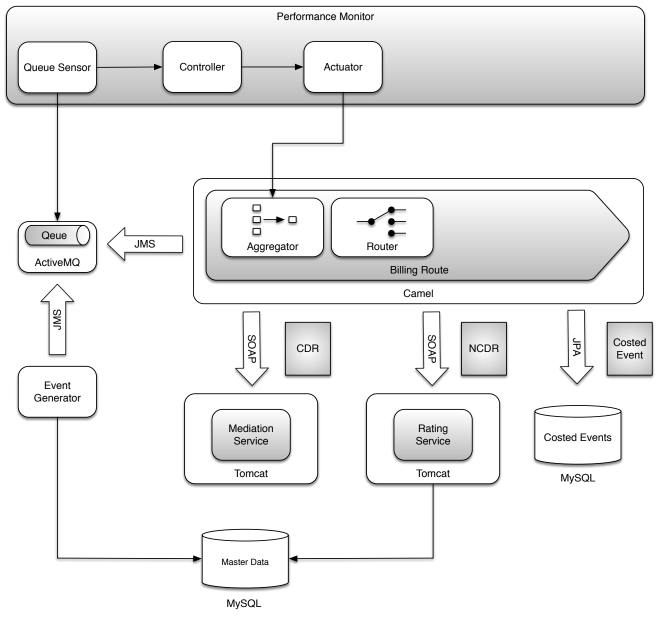


Figure 8. Components of the prototype system

Listing 2. Aggregator configuration in definition of BillingRoute

```
.aggregate(constant(true), new
    UsageEventsAggrationStrategy())
  .completionSize(header(completionSizeHeader)
      )
  .completionTimeout(completionTimeout)
  .parallelProcessing()
```

B. Feedback-Control Loop

Figure 9 shows the components of the feedback-control loop.

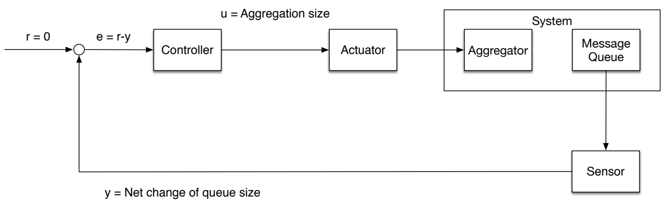


Figure 9. Components of the feedback-control loop

1) Sensor: The *JmxSensor* implements the *Sensor* interface (see Figure 10). It reads the current length of the input queue of the *ActiveMQ* server instance using JMX.

2) Controller: A *Controller* has to implement the *Controller* interface. The following implementations of the *Controller* interface have been implemented (see Figure 11):

- **BasicController**
  Implements a generic controller. The control strategy is provided by an implementation of the *ControllerStrategy*.
- **TestController**
  A controller used for testing the static behavior of the system.

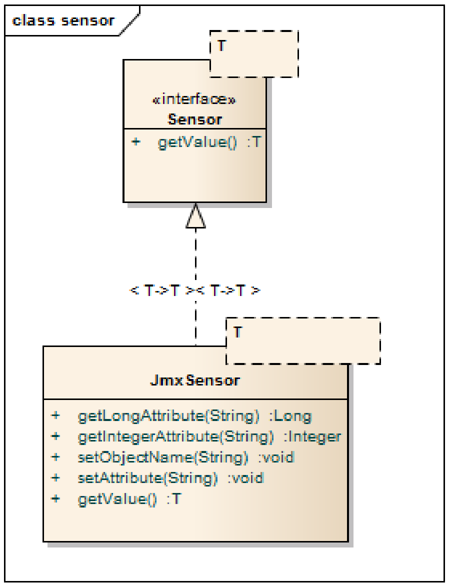

class sensor
T
«interface»
Sensor
+ getValue() :T
< T->T >< T->T >
T
JmxSensor
+ getLongAttribute(String) :Long
+ getIntegerAttribute(String) :Integer
+ setObjectName(String) :void
+ setAttribute(String) :void
+ getValue() :T

Figure 10. UML classdiagram showing the sensor classes

The strategy of the controller is implemented by a controller strategy which implements the *ControllerStrategy* interface.

Figure 12 shows the available implementations of the *ControllerStrategy*.

Listing 3 shows the implementation of the simple control strategy, as described in Section VI-D3:

- If the queue size increases, increase the aggregation size (line 10-13).
- Otherwise, do not change the aggregation size (line 22).
- Periodically decrease the aggregation size by one (line 17-20).

The controller uses two different timers depending on the previous action.

3) Actuator: The *AggregateSizeActuator* is responsible for setting the aggregation size of the *Aggregator* and is controlled by the *Controller* (see Figure 13).

The *AggregateSizeActuator* implements the *Actuator* interface. It sets the aggregation size (*completionSize*) by setting a specific header in the currently processed message.

4) Performance Monitor: The *Performance Monitor* manages the feedback-control loop by periodically calling the *Sensor* and updating the *Controller*. Additionally, it calculates the current throughput and end-to-end latency of the system using the *StatisticsService* (see Figure 14).

C. Load Generator

The *Load Generator* is used to generate the system load by generating events (Call Detail Records (CDRs)) and writing them to the input message queue of the system. It is implemented as a stand-alone Java program using a command-line interface.

The *DataGenerator* uses a *Poisson Process* to simulate the load of the system, which is commonly used to model

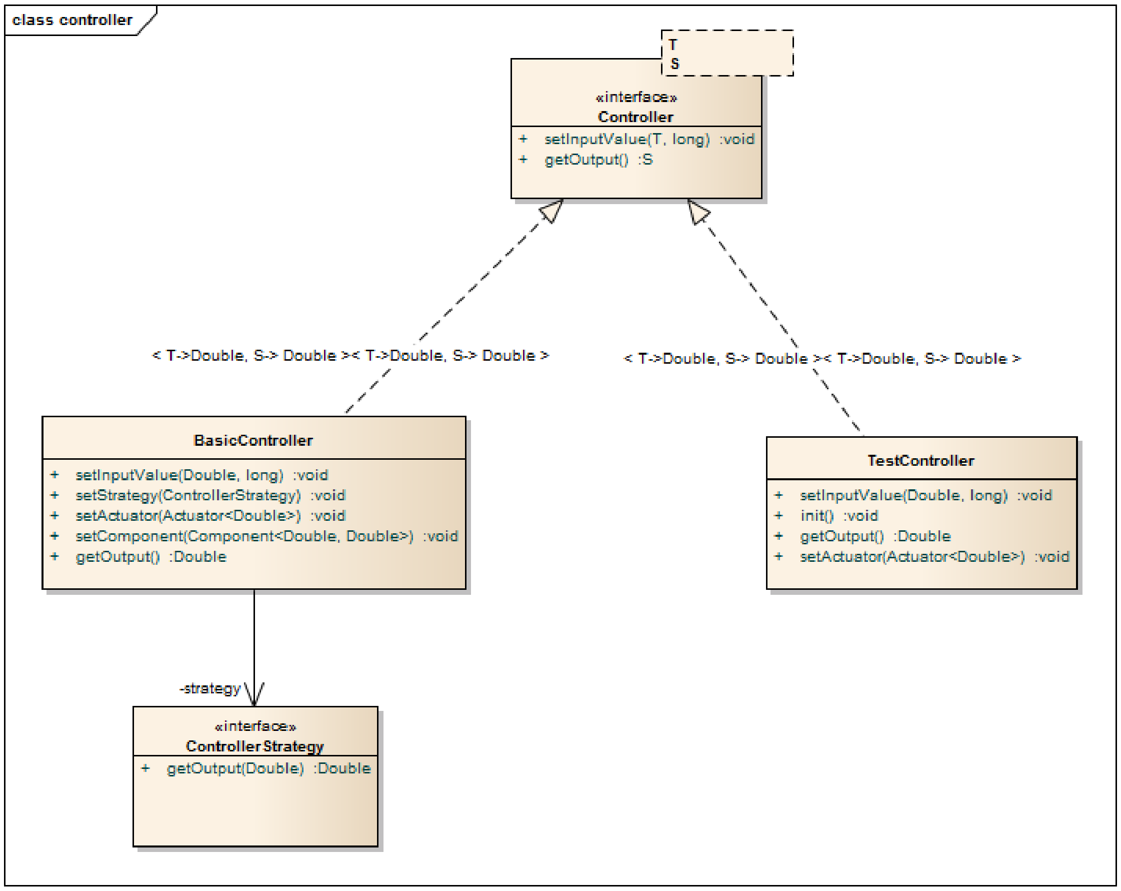

class controller
T
S
«interface»
Controller
+ setInputValue(T, long) :void
+ getOutput() :S
< T->Double, S-> Double >< T->Double, S-> Double >
< T->Double, S-> Double >< T->Double, S-> Double >
BasicController
+ setInputValue(Double, long) :void
+ setStrategy(ControllerStrategy) :void
+ setActuator(Actuator<Double>) :void
+ setComponent(Component<Double, Double>) :void
+ getOutput() :Double
TestController
+ setInputValue(Double, long) :void
+ init() :void
+ getOutput() :Double
+ setActuator(Actuator<Double>) :void
-strategy
«interface»
ControllerStrategy
+ getOutput(Double) :Double

Figure 11. UML classdiagram showing the controller classes

events that occur continuously and independently of each other with exponentially distributed inter-arrival times, e.g. to model requests on a web server [50] or telephone calls [51].

VIII. Evaluation

The prototype described in the previous section has been used to evaluate the general feasibility of the adaptive middleware.

A. Test Environment

The tests have been run on a development machine to decrease the development-build-deploy cycle, as described in Table V.

TABLE V
TEST ENVIRONMENT

Memory	3 GiB
CPU	Intel Core i5 M520 @ 2,40 GHz
Architecture	32-bit
Disk Drive	150 GB SSD
Operating System	Windows 7
Database	MySQL 5.5.24
Messaging Middleware	Apache ActiveMQ 5.6.0

B. Test Design

[52] defines a set of properties, that should be considered when designing feedback-control systems for computing systems, called the SASO properties (**S**table, **A**ccurate, **S**ettling times, **O**vershoot):

- **Stability**
  The system should provide a bounded output for any bounded input.
- **Accuracy**
  The measured output of the control system should converge to the reference input.
- **Settling time**
  The system should converge quickly to its steady state.
- **Overshoot**
  The system should achieve its objectives in a manner that does not overshoot.

C. Static Tests

To test the relationship between the input and output variables of the control-loop, aggregation size and change of queue size, the following static tests have been performed:

- The *TestController* has been configured to periodically increase the aggregation size after 100 time steps (1 time step equals 1 second).

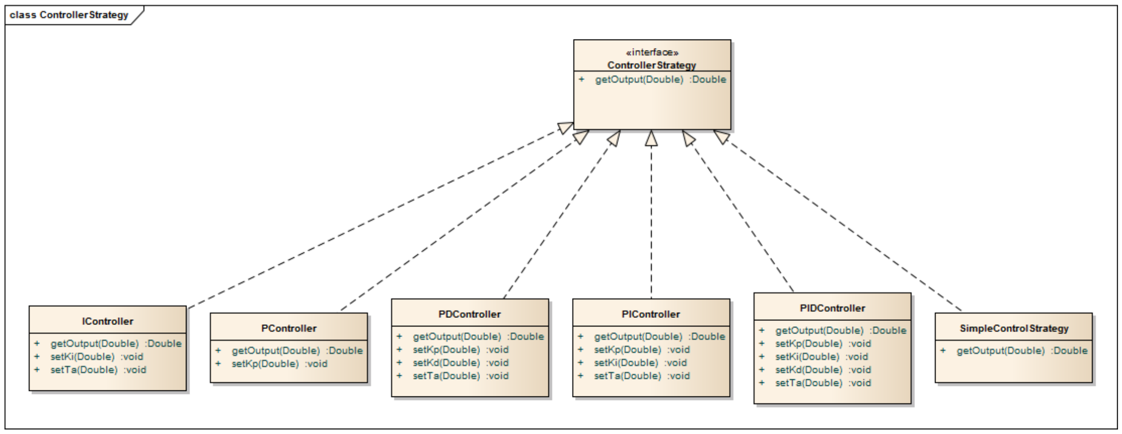


Figure 12. UML classdiagram showing the controller strategy classes

Listing 3. Implementation of the simple control strategy

```
public class SimpleControlStrategy
  implements ControllerStrategy {

  @Value("${simpleController.period1}")
  private int period1;
  @Value("${simpleController.period2}")
  private int period2;
  private int timer = 0;

  public Double getOutput(Double error) {
    if (error > 0) {
      timer = period1;
      return +1.0;
    }

    timer--;

    if (timer == 0) {
      timer = period2;
      return -1.0;
    }
    return 0.0;
  }
}
```

- The test has been repeated with different load of the system, that is, using different arrival rates for the *DataGenerator*.

Figure 15 shows the queue size of the system in relationship to the aggregation size, for different arrival rates.

- The system is not able to handle the load with an $aggregationsize < 5$ and an $arrivalrate = 50$. With an $aggregationsize \geq 5$, the system is able to process the events faster than they occur.
- With an $arrivalrate = 100$, the system is not able to handle the load with an $aggregationsize < 15$. With an $aggregationsize \geq 15$, the system is able to process the events faster than they occur.
- With an $arrivalrate = 150$, the system is not able to handle the load with an $aggregationsize < 25$. With an $aggregationsize \geq 25$, the system is able process the events faster than they occur.

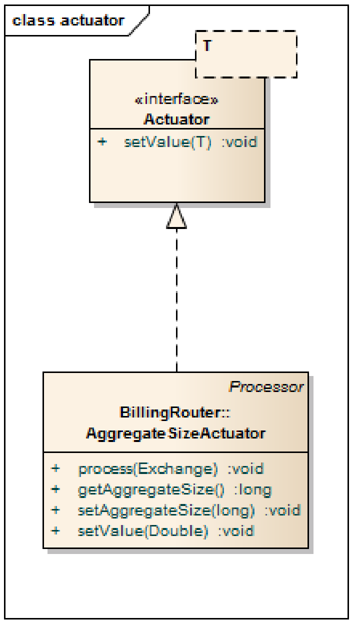


Figure 13. UML classdiagram showing the actuator classes

The change of queue size between each time step is shown in Figure 16.

D. Step Test

To measure the dynamic response of the system, the following step test has been performed:

- The *TestController* has been configured to increase the aggregation size from 1 to 50.
- Messages occur with an arrival rate of 150.

Figure 17 shows the result of the step test:

- With an aggregation size of 1, the system is not able to handle the load. The queue length is constantly increasing.
- When the aggregation size is set to 50 at timestep 100, the queue size is directly decreased, without a noticeable delay.

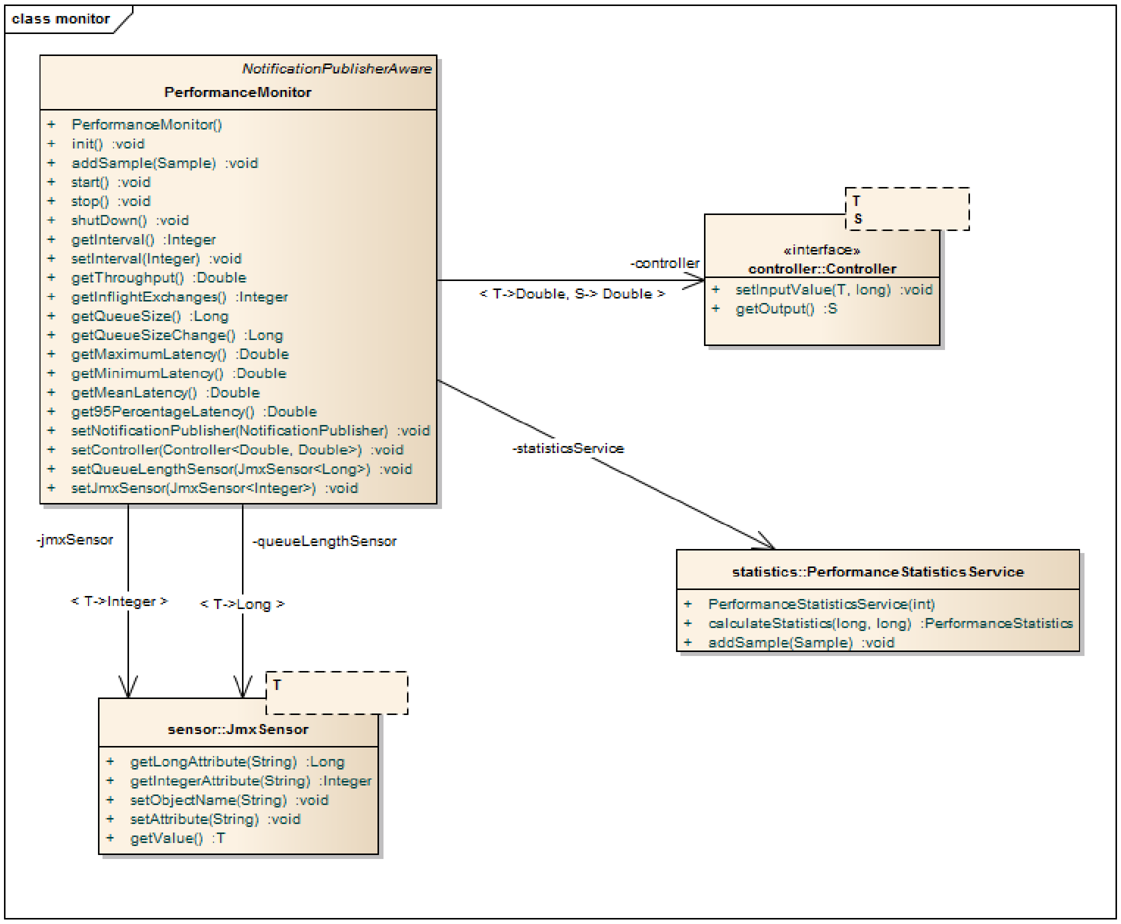

class monitor
NotificationPublisherAware
PerformanceMonitor
+ PerformanceMonitor()
+ init() :void
+ addSample(Sample) :void
+ start() :void
+ stop() :void
+ shutDown() :void
+ getInterval() :Integer
+ setInterval(Integer) :void
+ getThroughput() :Double
+ getInflightExchanges() :Integer
+ getQueueSize() :Long
+ getQueueSizeChange() :Long
+ getMaximumLatency() :Double
+ getMinimumLatency() :Double
+ getMeanLatency() :Double
+ get95PercentageLatency() :Double
+ setNotificationPublisher(NotificationPublisher) :void
+ setController(Controller<Double, Double>) :void
+ setQueueLengthSensor(JmxSensor<Long>) :void
+ setJmxSensor(JmxSensor<Integer>) :void
-controller
< T->Double, S-> Double >
T
S
«interface»
controller::Controller
+ setInputValue(T, long) :void
+ getOutput() :S
-statisticsService
statistics::PerformanceStatisticsService
+ PerformanceStatisticsService(int)
+ calculateStatistics(long, long) :PerformanceStatistics
+ addSample(Sample) :void
-jmxSensor
-queueLengthSensor
< T->Integer >
< T->Long >
T
sensor::JmxSensor
+ getLongAttribute(String) :Long
+ getIntegerAttribute(String) :Integer
+ setObjectName(String) :void
+ setAttribute(String) :void
+ getValue() :T

Figure 14. UML classdiagram showing the *PerformanceMonitor*

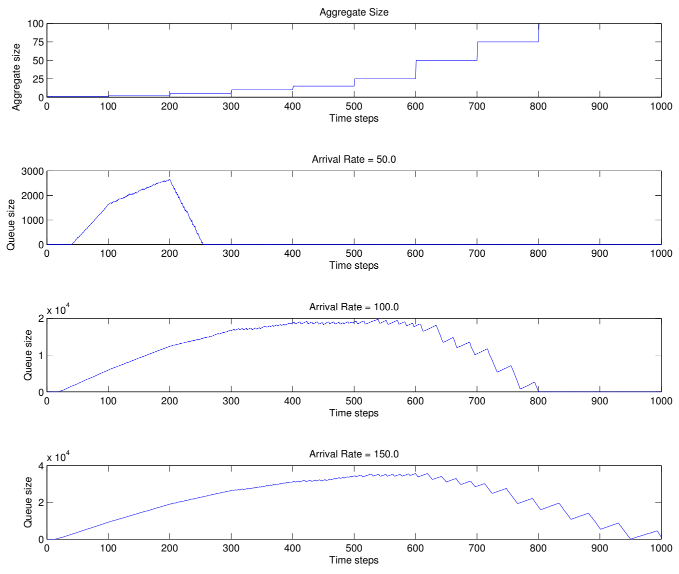


Figure 15. Static test: queue sizes

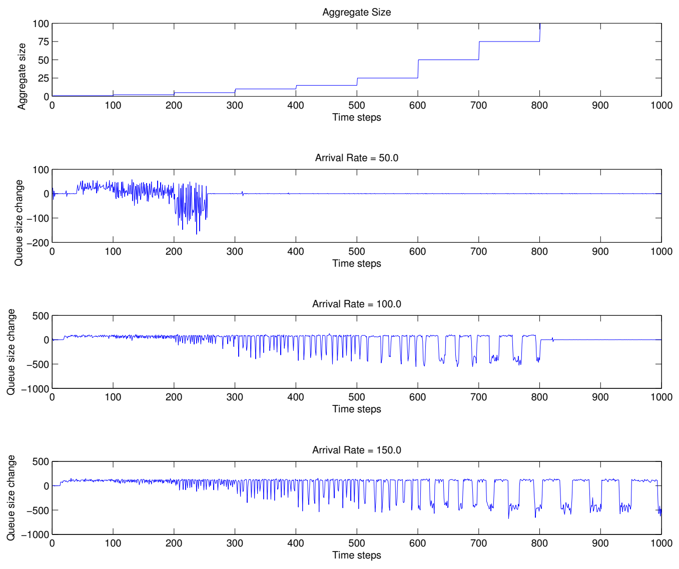


Figure 16. Static test: queue size changes

E. Controller Tests

The following test has been performed to evaluate the performance of the *Simple Controller*:

- Events are generated with an $arrival\ rate = 50.0$ for 100 time steps.
- At $timestep = 100$, the arrival rate is set to 150.0 for another 100 time steps.

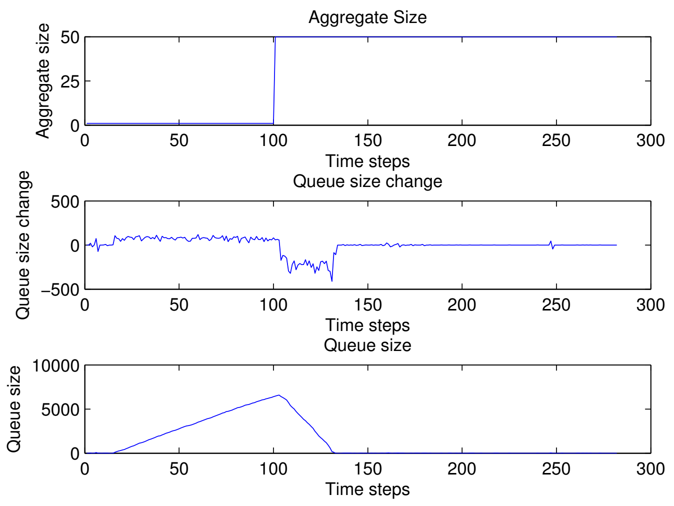


Figure 17. Step test

- At $timestep = 200$, the arrival rate is set back to 50.0.

Figure 18 shows the results of the test using the *Simple Control* strategy:

- The controller is reasonably able to control the size of the queue. At $timestep = 100$, it increases the aggregate size to a maximum value of 36.
- At $timestep = 200$, the controller starts to decrease the aggregation size. At $timestep = 375$, the aggregation size is back at 3.

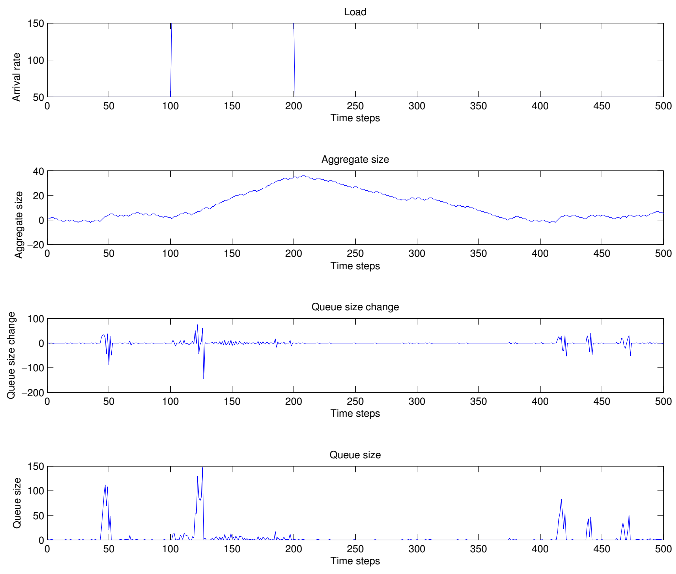


Figure 18. Simple control strategy

F. Results

Summarizing the results of the evaluation, the proposed concept for the adaptive middleware is a viable solution to optimize the end-to-end latency of data processing system. The results show that using a closed-feedback loop is a feasible technique for implementing the dynamic control of the aggregation size. Using the queue size change to measure the system load is also shown to be appropriate.

IX. Limitations

The research presented in this article has some limitations, that are summarized below:

- The services that implement the business functionality of the system need to be explicitly designed to support the run-time adaption between single-event and batch processing, as described in Section VI-A. Therefore, existing services need to be changed in order to be integrated into the system. This can pose a problem when using off-the-shelf services or Software as a Service (SaaS). The integration of such services has not been considered in this research.
- The services integrated by the prototype do not implement any further optimizations for batch processing. They use the same implementation for batch and single-event processing. Thus, the impact of batch optimizations has not been investigated. This was not necessary to show the performance improvements of message aggregation on the maximum throughput of the messaging prototype.
- The adaption mechanisms of the *Adaptive Middleware* only uses message aggregation and message routing, depending on the aggregation size. Other mechanisms such as dynamic service composition and selection and load balancing have not been investigated.
- The prototype of the *Adaptive Middleware* only uses a single message queue, the integrated services are called synchronous, using a request/response pattern. This design was chosen, to simplify the dynamics of the system. Thus, the impact of using multiple message queues has been investigated in the evaluation.
- The impact of different controller architectures has not been exhaustively analyzed and researched. Only two controller architectures have been implemented and evaluated. Other controller designs, such as fuzzy control, have not been investigated. Additionally, a formal analyzation of the feedback-control system has not been conducted, for example, by creating a model of the system. Instead, an empirical approach has been taken to evaluate the viability of the proposed solution.

X. Conclusion and Further Research

In this section, a novel concept of middleware for near-time processing of bulk data has been presented, which 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 message, the middleware routes a message to appropriate service endpoints, which are optimized for either single-event or batch processing.

Additionally, several design aspects have been described that should be taken into account when designing and implementing an adaptive system for bulk data processing, such as how to design the service interfaces, the integration and transport mechanisms, the error-handling and controller design.

The solution is based on standard middleware, messaging technologies and standards and fully preserves the benefits of an SOA and messaging middleware, such as:

- Loose coupling
- Remote communication
- Platform language Integration
- Asynchronous communication
- Reliable Communication

To evaluate the proposed middleware concepts, a prototype system has been developed. The tests show that the proposed middleware solution is viable and is able to optimize the end-to-end latency of a bulk data processing system for different load scenarios.

The next steps of this research are to further analyze the dynamics of the system and to optimize the controller.

During the implementation of the prototype of the adaptive middleware, it became apparent that the design and implementation of such a system differs from common approaches to implement enterprise software systems. Further research addresses a conceptual framework that guides the design, implementation and operation of a system for bulk data processing based on the adaptive middleware.

REFERENCES

[1] M. Swientek, B. Humm, U. Bleimann, and P. Dowland, "An Adaptive Middleware for Near-Time Processing of Bulk Data," in ADAPTIVE 2014, The Sixth International Conference on Adaptive and Self-Adaptive Systems and Applications, Venice, Italy, May 2014, pp. 37–41.

[2] J. Fleck, "A distributed near real-time billing environment," in Telecommunications Information Networking Architecture Conference Proceedings, 1999. TINA '99, 1999, pp. 142–148.

[3] J. Cryderman, "Is Real-Time Billing and Charging a Necessity?" Pipeline, vol. 7, no. 11, 2011.

[4] S. Conrad, W. Hasselbring, A. Koschel, and R. Tritsch, Enterprise Application Integration: Grundlagen, Konzepte, Entwurfsmuster, Praxisbeispiele. Elsevier, Spektrum, Akad. Verl., 2006.

[5] D. Chappell, Enterprise Service Bus. Sebastopol, CA, USA: O'Reilly Media, Inc., 2004.

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

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

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

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

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

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

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

[13] J. Tekli, E. Damiani, R. Chbeir, and G. Gianini, "Soap processing performance and enhancement," Services Computing, IEEE Transactions on, vol. 5, no. 3, 2012, pp. 387–403.

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

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

[16] D. Habich, S. Richly, S. Preissler, M. Grasselt, W. Lehner, and A. Maier, "BPEL-DT – Data-Aware Extension of BPEL to Support Data-Intensive Service Applications," Emerging Web Services Technology, vol. 2, 2007, pp. 111–128.

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

[18] L. Garces-Erice, "Building an enterprise service bus for real-time soa: A messaging middleware stack," in Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International, vol. 2, 2009, pp. 79–84.

[19] C. Xia and S. Song, "Research on real-time esb and its application in regional medical information exchange platform," in Biomedical Engineering and Informatics (BMEI), 2011 4th International Conference on, vol. 4, 2011, pp. 1933–1937.

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

[21] D. Bauer, L. Garces-Erice, S. Rooney, and P. Scotton, "Toward scalable real-time messaging," IBM Systems Journal, vol. 47, no. 2, 2008, pp. 237–250.

[22] J. Nagle, "Congestion control in ip/tcp internetworks," SIGCOMM Comput. Commun. Rev., vol. 14, no. 4, Oct. 1984, pp. 11–17. [Online]. Available: http://doi.acm.org/10.1145/1024908.1024910

[23] R. Friedman and R. V. Renesse, "Packing messages as a tool for boosting the performance of total ordering protocls," in Proceedings of the 6th IEEE International Symposium on High Performance Distributed Computing, ser. HPDC '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 233–.

[24] A. Bartoli, C. Calabrese, M. Prica, E. A. Di Muro, and A. Montresor, "Adaptive Message Packing for Group Communication Systems," 2003, pp. 912–925.

[25] P. Romano and M. Leonetti, "Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning," in Computing, Networking and Communications (ICNC), 2012 International Conference on, Jan 2012, pp. 786–792.

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

[27] R. Friedman and E. Hadad, "Adaptive batching for replicated servers," in Reliable Distributed Systems, 2006. SRDS '06. 25th IEEE Symposium on, 2006, pp. 311–320.

[28] H. A. Duran-Limon, G. S. Blair, and G. Coulson, "Adaptive Resource Management in Middleware: A Survey," IEEE Distributed Systems Online, vol. 5, no. 7, 2004, p. 1. [Online]. Available: http://portal.acm.org/ft_gateway.cfm?id=1018100&type=external&coll=ACM&dl=GUIDE&CFID=59338606&CFTOKEN=18253396

[29] B.-D. Lee, J. B. Weissman, and Y.-K. Nam, "Adaptive middleware supporting scalable performance for high-end network services," J. Netw. Comput. Appl., vol. 32, no. 3, 2009, pp. 510–524.

[30] D. Gmach, S. Krompass, A. Scholz, M. Wimmer, and A. Kemper, "Adaptive Quality of Service Management for Enterprise Services," ACM Trans. Web, vol. 2, no. 1, 2008, pp. 1–46.

[31] F. Irmert, T. Fischer, and K. Meyer-Wegener, "Runtime Adaptation in a Service-Oriented Component Model," in SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. New York, NY, USA: ACM, 2008, pp. 97–104.

[32] M. Leclercq, V. Quéma, and J.-B. Stefani, "DREAM: a Component Framework for the Construction of Resource-Aware, Reconfigurable MOMs," in ARM '04: Proceedings of the 3rd workshop on Adaptive and reflective middleware. New York, NY, USA: ACM, 2004, pp. 250–255.

[33] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The Case for Reflective Middleware," Commun. ACM, vol. 45, no. 6, 2002, pp. 33–38.

[34] R. Kazhamiakin, S. Benbernou, L. Baresi, P. Plebani, M. Uhlig, and O. Barais, "Adaptation of service-based systems," in Service Research Challenges and Solutions for the Future Internet, ser. Lecture Notes in Computer Science, M. Papazoglou, K. Pohl, M. Parkin, and A. Metzger, Eds. Springer Berlin Heidelberg, 2010, vol. 6500, pp. 117–156. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17599-2_5

[35] S.-H. Chang, H. J. La, J. S. Bae, W. Y. Jeon, and S. D. Kim, "Design of a dynamic composition handler for esb-based services," in e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on, Oct 2007, pp. 287–294.

[36] X. Bai, J. Xie, B. Chen, and S. Xiao, "Dresr: Dynamic routing in enterprise service bus," in e-Business Engineering, 2007. ICEBE 2007. IEEE International Conference on, Oct 2007, pp. 528–531.

[37] B. Wu, S. Liu, and L. Wu, "Dynamic reliable service routing in enterprise service bus," in Asia-Pacific Services Computing Conference, 2008. APSCC '08. IEEE, Dec 2008, pp. 349–354.

[38] G. Ziyaeva, E. Choi, and D. Min, "Content-based intelligent routing and message processing in enterprise service bus," in Convergence and Hybrid Information Technology, 2008. ICHIT '08. International Conference on, Aug 2008, pp. 245–249.

[39] A. Jongtaveesataporn and S. Takada, "Enhancing enterprise service bus capability for load balancing," W. Trans. on Comp., vol. 9, no. 3, Mar. 2010, pp. 299–308. [Online]. Available: http://dl.acm.org/citation.cfm?id=1852392.1852401

[40] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli, "Dynamic qos management and optimization in service-based systems," Software Engineering, IEEE Transactions on, vol. 37, no. 3, May 2011, pp. 387–409.

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

[42] "Amazon ec2 auto scaling," http://aws.amazon.com/autoscaling, [retrieved: March 2014].

[43] "Auto scaling on the google cloud platform," https://cloud.google.com/developers/articles/auto-scaling-on-the-google-cloud-platform, [retrieved: March 2014].

[44] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, Feedback Control of Computing Systems. John Wiley & Sons, 2004.

[45] R. K. Gullapalli, C. Muthusamy, and V. Babu, "Control systems application in java based enterprise and cloud environments–a survey," Journal of ACSA, 2011.

[46] G. Hohpe and B. Woolf, Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

[47] J. L. Hellerstein, "Challenges in control engineering of computing systems," in American Control Conference, 2004. Proceedings of the 2004, 2004, pp. 1970–1979.

[48] P. K. Janert, Feedback Control for Computer Systems. O'Reilly Media, Inc., 2013.

[49] "Apache Camel," http://camel.apache.org, 2014, [retrieved: July 2014].

[50] M. F. Arlitt and C. L. Williamson, "Internet Web servers: workload characterization and performance implications," IEEE/ACM Transactions on Networking (TON), vol. 5, no. 5, Oct. 1997, pp. 631–645.

[51] D. Willkomm, S. Machiraju, J. Bolot, and A. Wolisz, "Primary user behavior in cellular networks and implications for dynamic spectrum access," Communications Magazine, IEEE, vol. 47, no. 3, March 2009, pp. 88–95.

[52] T. Abdelzaher, Y. Diao, J. Hellerstein, C. Lu, and X. Zhu, "Introduction to Control Theory And Its Application to Computing Systems," in Performance Modeling and Engineering, Z. Liu and C. Xia, Eds. Springer US, 2008, pp. 185–215–215.