FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Adaptability in Distributed Stream Processing

Implementation and Evaluation using ESC

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Roland Kölbel
Matrikelnummer 0928067

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Dr. Benjamin Satzger

Wien, 01.10.2012

(Unterschrift Verfasserin) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Adaptability in Distributed Stream Processing

Implementation and Evaluation using ESC

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Roland Kölbel
Registration Number 0928067

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dr. Schahram Dustdar
Assistance: Dr. Benjamin Satzger

Vienna, 01.10.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Roland Kölbel
Schelleingasse 36, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasserin)

Abstract

The processing of large data sets is normally done using batch-oriented approaches. However, when confronted with the processing of live data streams and events, which are usually subject to high fluctuations in terms of data arrival, these batch-oriented approaches are not applicable.

For addressing these kinds of processing requirements, various stream processing engines have been developed. One of these engines is ESC [24], a cloud based stream processing engine designed for computations with real time demands written in Erlang [5]. In order to cope with increasing or decreasing computational needs, ESC is able to automatically scale by attaching or releasing nodes using information about the workload of the underlying machines. But, for preventing an overload of a node due to bursty data arrival or intensive computation, this approach alone is not sufficient. An adaptation technique is needed that, on the one hand, is able to identify unbalanced work distributions and take counteracting measures, and, on the other hand, is utilizing strategies of placing operators intelligently onto nodes, such that the chance of an overload situation becomes less likely.

The problem of mapping operators to nodes can be divided into two separate problems. First of all, the question on where to place operators initially is to be answered. Therefore, three different approaches have been implemented and analyzed. These include the random mapping of operators to nodes, the creation of operators on the currently least loaded node, as well as the mapping of an operator to a node, according to its unique name and considering the logic of the currently executed scenario.

Secondly, a strategy is needed, which is deciding when and where to move operators between nodes, in order to establish a balanced load distribution. Hence, two different load-balancing approaches have been implemented and analyzed. The first strategy balances the load by moving random workers from nodes with a high load to nodes with a lower load. In addition to that, the second strategy, which constitutes the main focus of this thesis, is derived from an existing solution to a problem of mapping tasks to processor nodes at run-time, which is called *Particles Approach* [29]. Therefore, a porting of the existing algorithm has been performed into the environment of distributed stream processing systems, together with an analysis of its effectiveness.

In order to compare the developed concepts with each other, a benchmarking application is required. Currently, the only available benchmark for stream processing systems is Linear Road [4], which simulates vehicles on expressways in a large metropolitan area. The developed methods are therefore checked using the Linear Road benchmark. Observed performance gains with different approaches are compared with each other and with the results of other stream computing engines.

Kurzfassung

Die Verarbeitung von großen Datenbeständen wird in der Regel mithilfe eines Batch-Verfahrens durchgeführt. Wenn jedoch die Verarbeitung von Datenströmen und Events in Echtzeit erforderlich ist, und das Volumen der ankommenden Daten hohen Fluktuationen unterliegt, können diese Batch-Verfahren nicht mehr angewendet werden.

Um diesen Anforderungen zu begegnen, wurden verschiedene Systeme zur Verarbeitung von Datenströmen entwickelt. Eines dieser Systeme ist ESC [24], ein cloud-basiertes System zur Durchführung von Echtzeit-Berechnungen auf Datenströmen geschrieben in Erlang [5]. Da die zur Berechnung erforderlichen Ressourcen ständigen Veränderungen unterliegen, ist ESC in der Lange, durch automatisches Hinzufügen oder Entfernen von Netzknoten zu skalieren. Die dafür notwendigen Informationen werden den Informationen zur Arbeitslast der zugrunde liegenden Maschinen entnommen.

Jedoch ist dieses Vorgehen alleine für die Verhinderung einer Überbelastung eines Netzknotens, aufgrund von stoßweise ankommenden Daten oder aufgrund von intensiven Berechnungen, nicht ausreichend. Eine Adaptionstechnik ist erforderlich, welche, auf der einen Seite, eine nicht ausbalancierte Lastverteilung erkennen kann und in der Lage ist, Gegenmaßnahmen zu treffen. Sowie, auf der anderen Seite, intelligente Strategien zur Platzierung von Operatoren auf Netzknoten nutzt, so dass die Wahrscheinlichkeit einer Überlastung deutlich reduziert wird.

Die Platzierung von Operatoren auf Netzknoten lässt sich in zwei separate Problemstellungen aufteilen. Zunächst ist die Frage zu beantworten, auf welchem Netzknoten ein Operator erzeugt werden soll. Diesbezüglich wurden drei unterschiedliche Strategien implementiert und analysiert. Dazu gehören die zufällige Zuordnung von Operatoren auf Netzknoten, die Erstellung von Operatoren auf dem am wenigsten ausgelasteten Netzknoten, sowie die Zuordnung eines Operatoren zu einem Netzknoten auf der Basis des eindeutigen Namens des Operators, als auch unter Einbeziehung des Graphen zum aktuell ausgeführten Szenario.

Weiterhin wird eine Strategie benötigt, welche eine Entscheidung trifft, wann und wohin Operatoren zu bewegen sind, um eine balancierte Lastverteilung innerhalb des Netzwerks herzustellen. Daher wurden zwei verschiedene Strategien implementiert und analysiert. Die erste Strategie balanciert die Lastverteilung durch die Verschiebung von zufällig ausgewählten Operatoren von Netzknoten mit hoher Last, zu Netzknoten mit geringer Last. Zusätzlich dazu wurde eine zweite Strategie entwickelt, welche den Hauptfokus dieser Arbeit bildet. Dieser Ansatz wurde abgeleitet von einer bereits existierenden Lösung zu dem Problem der Zuordnung von Aufgaben zu Prozessoren zur Laufzeit, bekannt unter dem Namen *Particles Approach* [29]. Demzufolge wurde der vorhandene Algorithmus in das Umfeld der Datenstrom-Analyse übertragen und, wenn notwendig, angepasst, sowie dessen Effektivität analysiert.

Zur Gegenüberstellung der entwickelten Konzepte und zur Belegung einer Leistungsverbesserung ist der Einsatz eines Benchmarks notwendig. Der einzige, aktuell verfügbare Benchmark für Systeme zur Datenstromanalyse ist Linear Road [4], in welchem Fahrzeuge auf Autobahnen in einer Großstadt simuliert werden. Die zu testende Applikation muss einen generierten Datenstrom verarbeiten, welcher für ein System zur Berechnung von Mautgebühren steht. Anschließend werden die errechneten Ergebnisse auf Korrektheit, sowie auf die Einhaltung maximaler Antwortzeiten, überprüft. Sämtliche entwickelte Methoden wurden daher mithilfe des Linear-Road-Benchmarks überprüft. Die beobachteten Leistungsveränderungen mit verschiedenen Strategien wurden miteinander, sowie mit den Ergebnissen anderer Systeme zur Datenstromanalyse, verglichen.

Contents

CHAPTER 1

Introduction

The amount of generated data online is increasing daily and is more and more taking the form of data streams, meaning a time ordered series of events or readings. Especially events generated from social networks are, on the one hand, rising in numbers each day, and on the other hand, becoming more and more important regarding their interpretation. Stream Data Management Systems are able to evaluate these live data streams in real time by executing continuous queries. In order to keep the latency of these systems low, to optimize their resource consumption and to ensure the best possible quality of service, adaptation to the current data stream load is necessary. Within this thesis, several adaptation methods have been developed, implemented and evaluated using the stream data management system ESC [24].

1.1 Background: Distributed Stream Processing

A large class of applications is emerging, in which data, generated by some external environment, is pushed asynchronously to servers that process this information. For this new application class conventional DBMS fall short, because stream-oriented systems are, in contrast to DBMS, predominantly geographically distributed and their distribution offers scalable load management and higher availability. These new applications include, for example, sensor networks, location-tracking services, fabrication line management, network management and social networks as well. What characterizes these applications is their need to process high-volume data streams in a timely and responsive fashion. Therefore, these applications are typically called "stream-based" applications. The difference between this new class of applications and traditional DBMS is that the architecture of current databases assumes a pull-based model of data access. This means that when a user wants to have access to the data, he submits a query to the system and an answer is returned. In contrast to that, stream-based applications invert the traditional data management model by assuming users to be passive and the data management system to be active. The data is pushed to the system, which must evaluate a given set of queries in response, to detect events. Query answers are then pushed to the waiting user or to another application [31].

1.2 Background: Adaptability Techniques

Many stream management systems are inherently distributed and therefore large and unpredictable in terms of system parameters. The tuning of system performance for these systems turns out to be a very challenging task, but vital, as these systems process live events and their latency must be as low as possible. As a consequence, collecting accurate statistics of system parameters from all involved nodes at runtime would be necessary in order to make the right decisions regarding the improvement of system performance and system efficiency. These system parameters include properties of the data stream itself (data arrival rates, value distribution, etc.), the load of the processing server, the network transfer rate and the network transfer delay. Predictions about these system parameters are very hard and, in addition to their unpredictability, they may evolve over time. Due to these complicating factors, the initial system setup most likely will result in unsatisfying system performance. To be less vulnerable under these circumstances, the ability to adapt itself to changing system parameters without human intervention is absolutely vital for stream processing applications.

In order to keep the latency low, despite unforeseeable data rates and possible bursty data arrival, stream processing systems must have plans to adapt accordingly. Several techniques already exist dealing with the question of adaptation in distributed environments. A small overview of discovered techniques together with a short description is presented in Table 1.1. Chapter 2.2 then gives a more detailed description of adaptation techniques and how they work in particular.

Technique	**Description**
Adaptation by load shedding	When an overload is detected as a result of static or dynamic analysis, the incoming event tuples are reduced. One way to do that is dropping tuples at random points in the network in an entirely uncontrolled manner. Another way of doing load shedding is to drop tuples depending on the importance of the packets contents by the definition of certain quality of service information [2].
Adaptation by load balancing	Whenever the system shows a degradation in latency, it is possible that only a small subset of all available nodes is responsible for the loss in performance. Therefore, actions can be taken to distribute the load across all nodes in a more efficient manner.
Adaptation by releasing or attaching processing resources	Especially, when a system is deployed into a cloud environment, the availability of processing resources is not an issue. Nevertheless, processing resources and energy resources should not be wasted by having every available processing resource enabled and dedicated to the stream processing system, when it is not needed. The stream processing system should detect automatically when it can release or attach processing resources, and do so whenever feasible.

Table 1.1: Overview Adaptability Techniques

1.3 Contribution

Within this thesis, several adaptation techniques have been implemented, analyzed and compared with each other. Each technique deals with the problem of how to map operators to nodes in an efficient manner, which can be divided into two separate problems. First of all, the question on where to place operators initially is to be answered. Therefore, three different approaches have been implemented and analyzed. These include the random mapping of operators to nodes, the creation of operators on the currently least loaded node, as well as the mapping of an operator to a node, according to its unique name and considering the logic of the currently executed scenario.

Secondly, a strategy is needed, which is deciding when and where to move operators between nodes, in order to establish a balanced load distribution. Hence, two different load-balancing approaches have been implemented and analyzed. The first strategy balances the load by moving random workers from nodes with a high load to nodes with a lower load. In addition to that, the second strategy, which constitutes the main focus of this thesis, is derived from an existing solution to a problem of mapping tasks to processor nodes at run-time, which is called *Particles Approach* [29]. Therefore, a porting of the existing algorithm has been performed into the environment of distributed stream processing systems, together with an analysis of its effectiveness.

In order to compare the developed concepts with each other, and to prove a gain in the processing performance of the system, a benchmarking application is required. Currently, the only available benchmark for stream processing systems is Linear Road [4], which simulates vehicles on expressways in a large metropolitan area. An incoming data stream, simulating a tolling system for vehicles, needs to be processed by the tested application. Afterwards, the benchmark verifies that every result arrived at the correct time and issues a rating. The developed methods are therefore checked using the Linear Road benchmark. Observed performance gains with different approaches are compared with each other and with the results of other stream computing engines.

CHAPTER 2

State of the Art

Over the last ten years, distributed stream processing has found its way from academia into the industry. More and more use cases arise as the flood of daily generated data is increasing rapidly and its evaluation is getting more and more important. The following chapters include an overview of the concepts and techniques that are currently in use in the productive environment regarding distributed stream processing.

2.1 Distributed Stream Processing

The main aspect of stream processing is its possibility to filter data rapidly by the help of Continuous Queries (CQ). In traditional systems, the arrival of queries initiates access to a stored collection of data. On the contrary, in a stream processing system, the arrival of data initiates access to a stored collection of queries. Arriving data, over which Continuous Queries are executed, can be effectively infinite. Therefore every used query operator must be non-blocking and must continuously return incremental results [9].

TelegraphCQ

TelegraphCQ is a Data Stream Management System developed by UC Berkeley. It contains a suite of novel technologies for continuously adaptive query processing [9]. The Telegraph project has been initiated at UC Berkeley at the beginning of the year 2000 with the goal of developing an adaptive dataflow architecture for supporting a wide variety of data intensive, networked applications [9]. Initially, TelegraphCQ was used to support Federated Facts and Figures (FFF), a query system for deep-web data [25]. The implementation is written in C/C++ and heavily leverages the PostgreSQL code base. TelegraphCQ offers the evaluation of several continuous queries over several data streams with support for windowed queries using a basic version of the SQL dialect. The architecture of TelegraphCQ is shown in Figure 2.1.

In order to receive queries from the clients, a listener interface is available, which accepts multiple continuous queries and adds them dynamically to the running executor. When a query

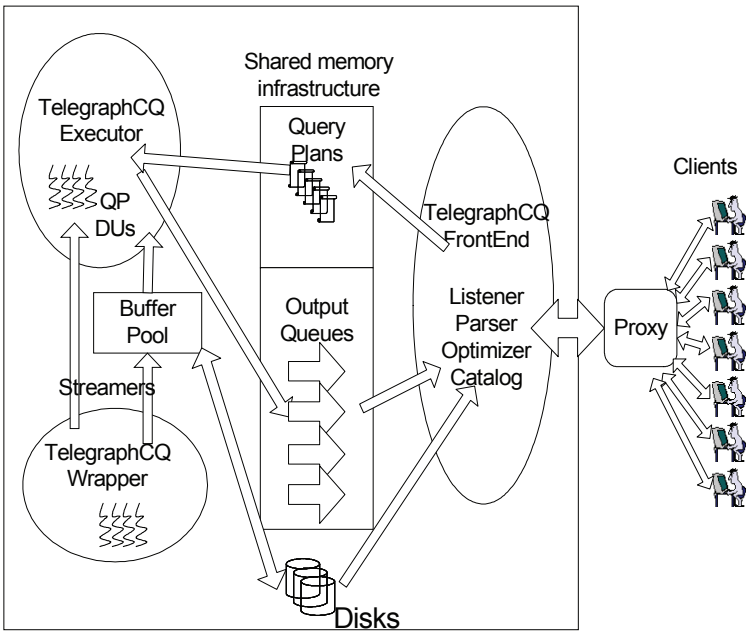


Figure 2.1: Architecture of TelegraphCQ (taken from [9])

is received, the server parses, analyzes and optimizes the query into an adaptive query plan. These plans are dynamically folded into the running queries in the executor. The results of these queries are then placed in client-specific output queues. Another listener picks up results from the output queues and sends the results back to the client applications [9].

The TelegraphCQ project has been closed in 2006, but several spin-offs, like for example Truviso [10], are still under active development.

Aurora

Aurora is a single-site stream engine developed at Brandeis University, Brown University and MIT [31]. The development of Aurora began 2002 and ended 2006 with the start of its successor project, “Borealis”, which is presented below. Aurora evaluates data-flows and, hence, applies the boxes and arrows paradigm, which means that incoming tuples flow through a loop-free, directed graph of processing operations. These operations currently contain seven primitive operators, like for example “filter”, “aggregate”, “union” and “resample” [2].

Further on, Aurora allows the definition of quality of service requirements by the use of quality of service graphs, that specify the utility of the output in terms of several performance related and quality related attributes [2]. An image of the basic architecture of Aurora is presented in Figure 2.2.

In contrast to the systems presented previously, distributed stream processing systems focus more on distribution and on the balancing of load within the node network. When a system is

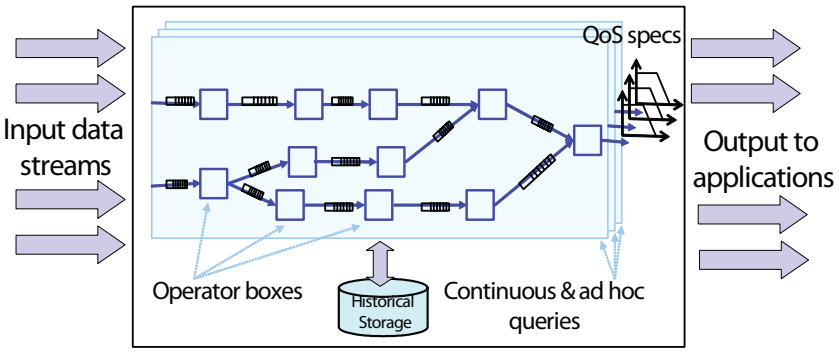


Figure 2.2: Aurora system model (taken from [8])

confronted with very high and fluctuating load, having an effective distribution and load balancing strategy is essential, especially in stream processing environments. The more a system is able to scale, when subjected to heavier load, the better. A couple of representative systems, that make use of distribution, is presented in the following chapters together with a short description of their applied strategies.

Borealis

Borealis is a second-generation distributed stream processing engine, that is being developed at Brandeis University, Brown University and MIT as the successor of the Aurora stream processing system. In order to add distribution capabilities to the already present stream processing capabilities of Aurora, Borealis inherits the core stream processing functionality from Aurora and the distribution functionality from Medusa, a scalable distributed stream processing system developed at MIT in 2003 [19].

As Aurora, Borealis supports continuous queries, which can be seen as one framework of operators, whose processing is distributed to multiple sites [1]. The components of one Borealis node and its exposed interfaces are depicted in Figure 2.3. Together with the features inherited by Aurora, Borealis also supports the dynamic revision of query results, dynamic query modifications as well as scalable optimization [1].

A Borealis application is a single connected diagram of processing boxes deployed on a network of N servers, which can be referred to as a site [1]. The components shown in Figure 2.4 continuously optimize the allocation of query network fragments to processing sites. These components can be categorized into “Monitors” and “Optimizers”, and can be described as follows [1].

- **Monitors:** There are local monitor types and global monitor types. Local monitors run at each site and produce a collection of local statistics, which they forward periodically to the end-point monitor. The end-point monitor runs at every site and evaluates the quality of service for every output message and keeps the quality of service statistics.

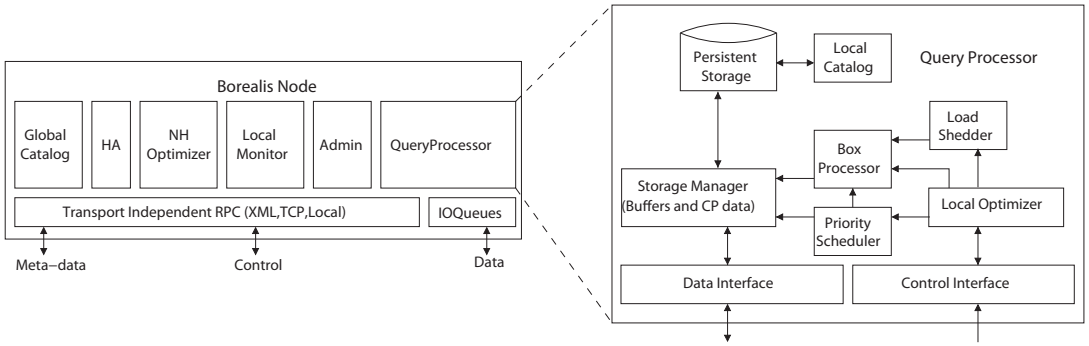


Figure 2.3: Borealis Architecture (taken from [1])

- **Optimizers:** A local optimizer runs at every site and schedules messages as well as sheds load, if necessary. Another optimizer, called the neighbourhood optimizer, balances the resources at a site with those of its immediate neighbours. The global optimizer accepts information from the end-point monitors and makes global optimization decisions.

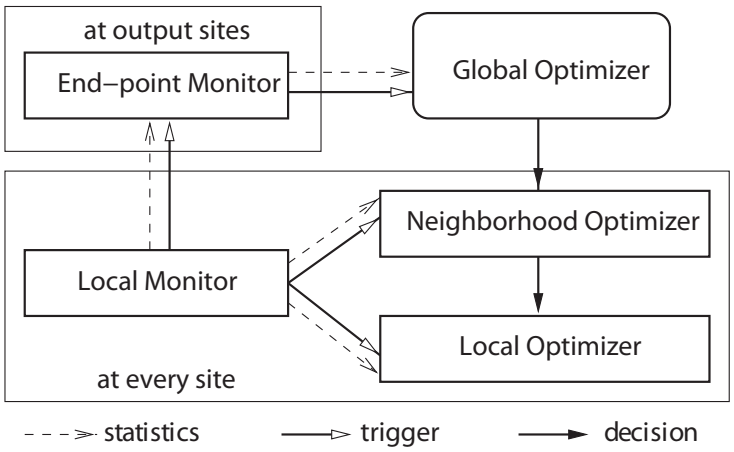


Figure 2.4: Borealis Optimizer Components (taken from [1])

System S and SPADE: The System S Declarative Stream Processing Engine

System S is a distributed stream processing middleware under development at IBM T. J. Watson Research Centre [14]. The runtime of System S can be scaled from one node to thousands of computer nodes. Queries are executed as Data-Flow Graphs, consisting of a set of processing elements connected by streams. The relevant parts of SPADE inside the System S are shown in Figure 2.5.

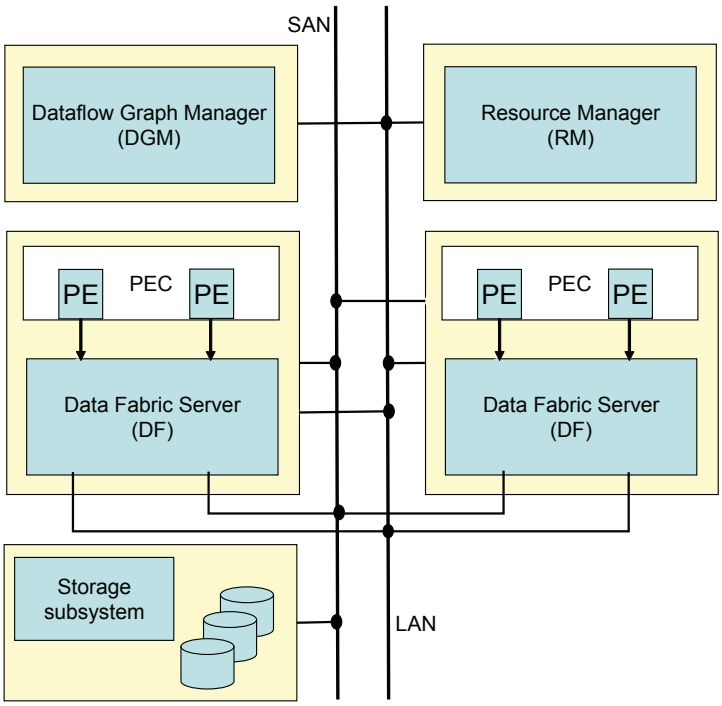


Figure 2.5: Stream processing core of System S (taken from [14])

The following features are provided by SPADE for the System S middleware [14].

- An intermediate language for flexible composition of parallel and distributed data flow graphs. The language includes standard operators like “Functor”, “Aggregate”, “Join”, “Sort”, “Barrier”, “Punctor”, “Split” and “Delay”
- A toolkit of type-generic, built-in stream processing operators
- A rich set of stream adapters to ingest/publish data from/to outside sources

In order to run stream processing applications on the System Processing Core (SPC) of System S, SPADE employs a code generation framework that transforms applications into the required format. In addition to these features, SPADE introduces performance optimization and scalability to System S applications by applying three optimization strategies, which are created by SPADE's code generation framework [14]:

- **Operator Grouping:** The problem of finding the best operator-to-PE mapping for balancing load (an example is illustrated in Figure 2.6) is solved in SPADE by the use of an optimization partitioner. By having information about the CPU load and network traffic statistics for the data-flow graph, the partitioner aims at minimizing the total inter-PE communication.

- **Execution Model Optimization:** Build-in operators could generate multi-threaded code, which runs on many cores in parallel. Assuming multi-threaded code for each operation, the problem is to decide how to efficiently distribute threads to operators within a PE.

- **Vectorized Processing Optimization:** The vectorized operations on list types get accelerated through Single-Instruction Multiple-Data (SIMD) operations available in most modern processers. SPADE makes use of Streaming SIMD Extensions (SSE) on the Intel processors to accelerate the basic arithmetic operations on list types.

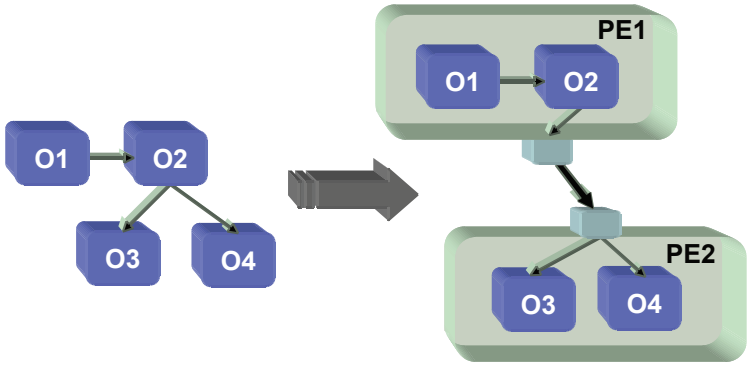


Figure 2.6: Example Operator-to-PE mapping in SPADE (taken from [14])

SCSQ

SCSQ allows the processing of stream queries on supercomputers and has been developed at Uppsala University in 2006 [32]. Its initial purpose was to process data streams from a radio telescope with data rates of several terabits per second [17]. The name SCSQ stands for Super Computer Stream Query Processor, pronounced "cisqueue", and the system is running on the BlueGene IBM cluster (see also Figure 2.7).

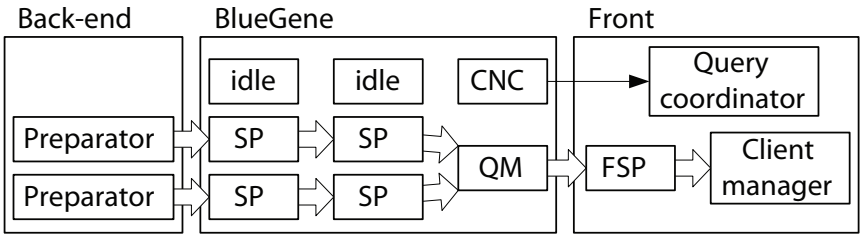


Figure 2.7: Components of SCSQ (taken from [32])

The scaling inside the SCSQ system happens by dynamically incorporating more computational resources as the amount of data grows. As the system is operating inside of a cluster, the adding and releasing of resources is cheap in comparison with, for example, distributed environments inside local area networks.

In order to reduce the volume of the resulting stream in realtime, continuous queries filter and transform the stream to identify events. For an efficient filtering and transforming of the streams before merging and joining them, SCSQ supports sub-queries parameterized by stream identifiers, which execute in parallel on different nodes. The executed continuous queries are specified declaratively by the use of a query language similar to SQL, extended with streaming and vector processing operators [33].

The implementation of SCSQ is based on AMOS II (Active Mediator Object System), which has been modified to allow the execution of continuous queries. Currently, the SCSQ project is still under active development and reaches the highest scores in Linear Road [33].

Yahoo S4

Yahoo S4 is a distributed platform, that allows programmers to develop applications for processing continuous, unbounded streams of data [21]. Every event inside the system has a key assigned to it and is routed with affinity to processing elements, which consume the events and either emit new events or publish results [21].

Originally, Yahoo S4 has been developed in order to render the most relevant ads in an optimal position on a “search and results” page based on a data stream containing user preferences, geographic location, prior queries or prior clicks. The system follows the actors model [3] and makes use of Apache ZooKeeper [28] in order to perform cluster management, that can be shared by many systems in the data center. An example of a query inside of Yahoo S4 is given in Figure 2.8.

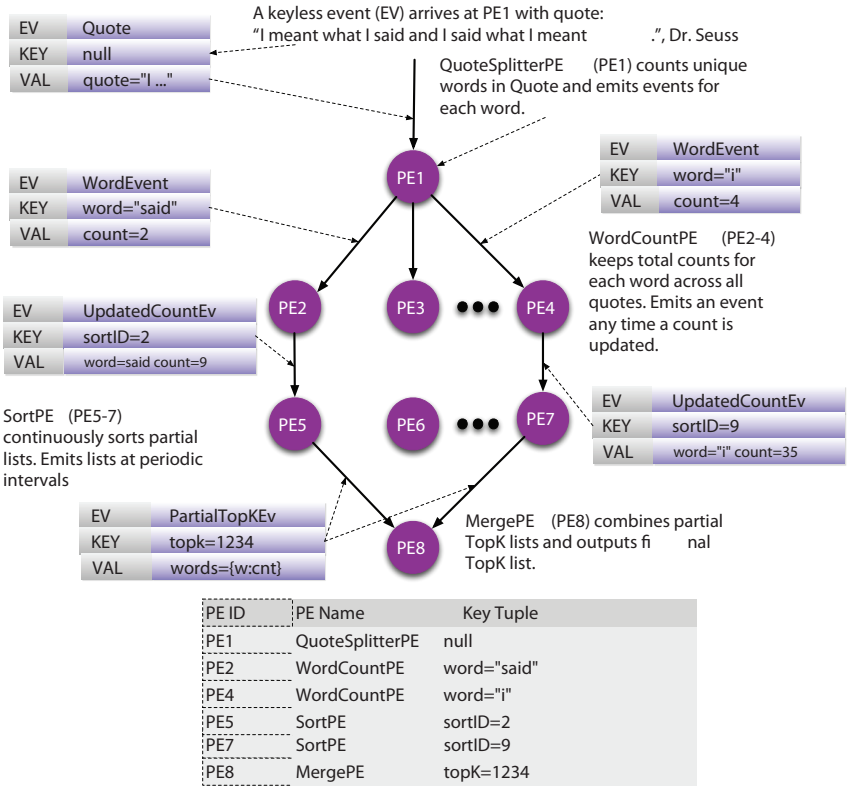


Figure 2.8: Word Count Example in Yahoo S4 (taken from [21])

Complex Event Processing in Commercial Enterprises

Complex event processing is used especially inside distributed computing and information systems, which are systems that automate the operations of commercial enterprises. With the advancing of technology and the growth of the Internet, these distributed information processing systems grew beyond the single enterprise, across the boundaries between enterprises. Information is no longer limited to the domain of one enterprise, but shared between different enterprises and, therefore, the foundation for trading partnerships and the automation of business collaborations. When viewed at a macro level, all these various enterprises and organisations are components of the system, which communicate through the use of networks. Messages, or "events", flow across these networks between these enterprises. The components react to the events they receive and issue new events that are sent to other components. These systems are so called "event driven" – they live or die based upon the message flowing across their IT networks. As for the year 2001, 5.000 till 10.000 messages per second flowed through a single large brokerage house's information technology layer, and soon that number will be even higher [18].

The main problem of these so called "enterprise systems" is the huge amount of generated events, often up to zillions of events per hour or per day. Currently, there is no technology

capable of viewing these events and activities that are going on inside these systems in ways that humans can understand. Enterprises are investing huge amounts of money into the development of tools that are able to monitor events in the basic networks that carry information. The challenge in developing these tools is to answer questions about events that are not simply low-level network activities, but are high-level activities related to what the systems are intended to achieve – so called business-level or strategic-level events. By the use of these tools it should be possible to answer questions like "What caused our trading system to sell automobiles to a customer in Texas?" or "What is causing the system to fail to execute this trading agreement?". These questions are about complex events, which are build out of lots of simpler events [18].

In order to do complex event processing, an event processing technology is needed that solves or tackles the following related problems [18]:

- Monitor events at every level in IT systems
- Detect complex patterns of events
- Trace causal relationships between events in real time
- Take appropriate action when patterns of events of interest or concern are detected
- Modify our monitoring and action strategies in real time

What is an event?

According to [18], an event is an object that is a record of an activity in a system. An event may be related to other events and has three aspects:

- **Form:** The form of an event is an object, which may have particular attributes or data components. These can include, for example, the time period of the activity.
- **Significance:** As an event signifies an activity, the form of an event usually contains data describing the activity it signifies.
- **Relativity:** An event is related to other events by time, causality and aggregation, which is called its relativity.

Events are often confused with messages by assuming "An event is just a message". This is due to the fact that message generation is a common way of generating events that signify activities. The difference is that events also have significance and relativity. Additionally, event processing must deal with relationships between events in contrast to message processing [18].

Currently, enterprises are becoming, or have already become, event-driven, autonomous, information processing systems, so called electronic enterprises [18]. An example for an event-driven business process is given in Figure 2.9. The workflow "ProcessOrder" is initiated inside the enterprise by an event, which is stating that a customer placed an order for a product. At the end of the workflow, the enterprise initiates the activity of billing the customer. In carrying out this simple linear workflow, the enterprise boundaries are crossed many times by outgoing and incoming events. Each compiled rule is trigged by an event from one step of the process and generates a new event that initiates the next step.

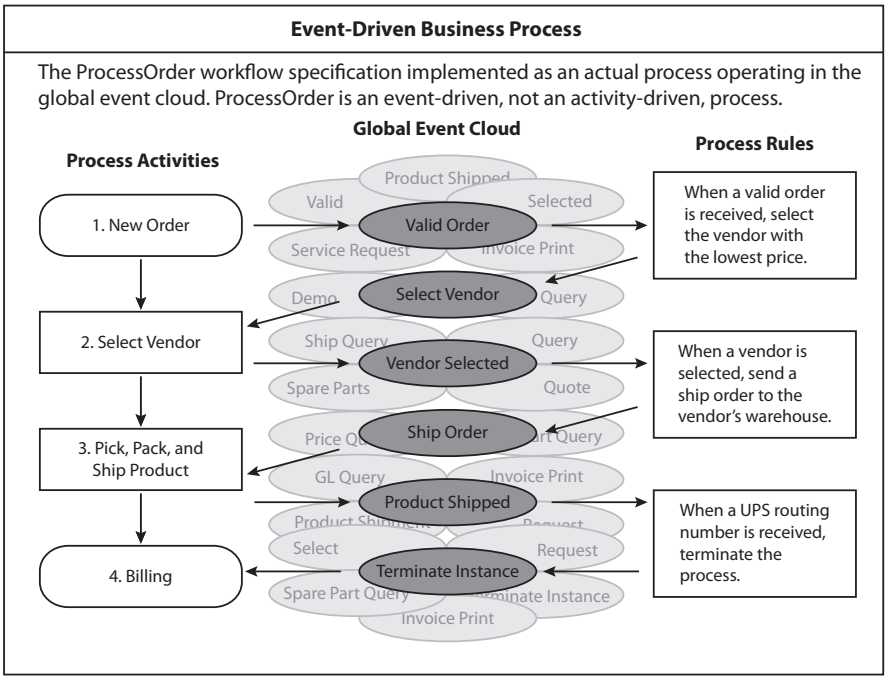


Figure 2.9: Example enterprise operation in the global event cloud (taken from [18])

2.2 Adaptation to Load Changes

Stream processing systems evaluate events that arrive at an unpredictable rate. When these input rates exceed the machine's capacity, the system will become overloaded and the latency will deteriorate. These overload situations are usually unforeseen and an immediate reaction is vital, adapting the system's capacity to the increased load by adding more resources or by distributing computation to multiple nodes. If these approaches are not possible, not feasible or not economically meaningful, the system must shed load and thus degrade the quality of the answer in order to guarantee latency.

The addition of resources in order to compensate shortages in processing power is certainly the most straightforward way of addressing the problem. Due to the fact that adding new processing power is, in most scenarios, not economically feasible and generally takes too much time until the new components are operable, this concept is not further discussed throughout this thesis. The strategy of load balancing, on the other hand, is one of the main topics of this thesis. Current approaches are discussed in depth in Chapter 3. In the remaining part of this chapter, techniques that deal with the strategy of load shedding are described in detail.

Load shedding is the process of dropping excess load from the system and should only be applied when no other approach, which does not degrade the answer, is promising to restore normal operation. In an overload situation the system will shed load, thus degrading the answer in order to improve the observed latency. For doing effective load shedding, the questions, when to shed load, where to shed load and how much load to shed, must be answered.

One technique for doing load shedding, that has been proposed by Tatbul et al. [26], suggests the dynamic insertion or the dynamic removal of "drop" operators into or from query plans, as required by the current load. An example of how this adding and removing of "drop" operators could look like, is illustrated in Figure 2.10. According to this technique, two types of "drop" operators exist. The first type drops a fraction of the incoming tuples in a randomized fashion, whereas the other type drops tuples based on the importance of their content.

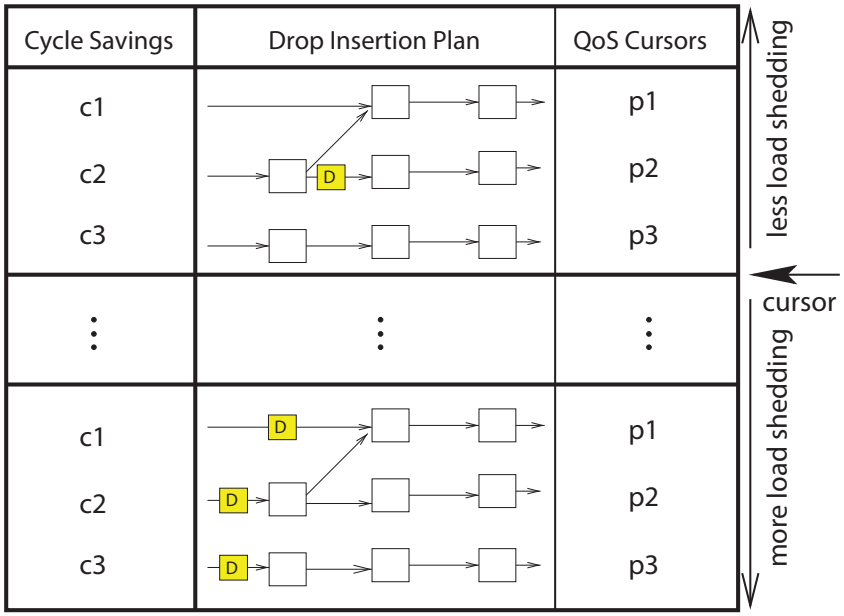


Figure 2.10: Load Shedding Road Map (taken from [26])

Another approach suggested by Tatbul et al. [27], takes into account that resource management decisions at any server node will affect the characteristics of the workload received by its children. Because of this load dependency between nodes, a given node must figure out the effect of its load shedding actions on the load levels of its descendant nodes. The approach continues by modelling the distributed load shedding challenge as a linear optimization problem, which is solved, on the one hand, with the help of a solver as a centralized solution, and, on the other hand, as a distributed approach based on metadata aggregation and propagation.

2.3 Conclusion

Complex events and their processing are becoming more and more important, especially in enterprise environments. A huge variety of stream processing systems have already been developed and are in productive use today. In order to adapt to fluctuating loads, these systems apply certain strategies for which some have been described in detail. New techniques and approaches, which are subject to current research and deal more with the load balancing aspect of distributed stream processing systems, are described in the following chapter.

CHAPTER 3

Related Work

Trends show that IT is more and more transformed into large integrated service networks with growing complexity, which results in potential ineffectiveness and insufficient manageability [15]. In order to regain efficiency, and thereby improve manageability, strategies and algorithms are needed that address these new conditions. On the one hand, the question of where to put services, the so called *Placement Problem*, needs to be answered by these approaches. On the other hand, these approaches need to propose a strategy on how to balance the load emitted by these services inside the service network, the so called *Load Balancing Problem*. In the following chapters, approaches are presented, that neither require central control nor complete information about the system itself, and which focus on how the adaptation problem can be solved.

3.1 Nature Inspired Algorithms

Biologically inspired algorithms use behavioural patterns from the field of biology and have been adapted in order to be applied in the field of computer science. In doing so, the results of millions of years of evolution guide as a model for optimizations to solve related problems, like the problem of adaptability in computer networks. In the following chapters, three algorithms originating from the field of nature are presented. The first algorithm described is inspired by ant colonies, the second algorithm simulates a hormone system and the third described algorithm replicates cell transformation and locomotion.

Ant Colony Algorithm

Ant Colony Optimization [11] is a cooperative meta-heuristic that is being successfully applied to various combinatorial optimization problems [22]. The desirable feature of ant colonies is their ability to find the shortest path from their nest to a food source in a relatively short time, without any initial knowledge of the surrounding environment and in a completely decentralized way. In order to achieve these vital characteristics, ants communicate in an indirect manner, which is called stigmergy. First of all, ants deposit traces of pheromone on their trail, which makes this trail more attractive to other ants. Afterwards, an evaporation of pheromones makes a path less attractive. When alternative trails are chosen randomly at the beginning, the pheromone level of a path is inverse proportional to the path's length with high probability [22]. This behaviour of ants is illustrated in Figure 3.1, where an obstacle is placed into a real ant trail and the shortest path C is chosen due to a much higher amount of pheromones [11].

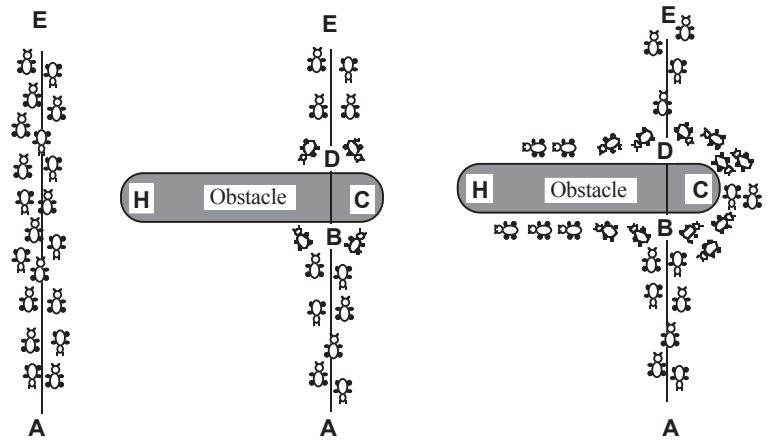


Figure 3.1: Ant Colony Algorithm (taken from [11])

For the question on how this Ant Colony Optimization can be transferred to solve the problem of adaptability, several suggestions have been made. Rammig et al. describe an approach to use Ant Colony Optimization for service migration in sensor networks. The desired goal, according to the paper, is to dynamically find a mapping of services to nodes, such that the global communication costs between services and application tasks, requesting these services, are minimal. Within the proposed algorithm, services are the equivalent of food sources, calls made by the requesters represent the ants, whereas the requesters represent the formicaries. Wireless links between the nodes constitute the paths, which the ants can use for their walks. While requests are being routed to the destination service, they leave pheromone on the nodes, which "evaporate" over time, meaning they are deleted after their timer expired. By using this mapping, requests choose with high probability the shortest path after a certain amount of time, without central control.

Another approach on how Ant Colony Optimization can be used to assign services to nodes has been proposed by Graupner et al. [15]. For each service a service manager M_s is instantiated,

which creates multiple ants and sends them out to the network. Each ant is equipped with a service list containing the service, the services cooperating with it and their current resource requirements. The ant travels from one server to another choosing the servers along its path based on a probability computed locally. In each step, one of the services from the list is assigned to the current server, as shown in Figure 3.2. On each server the ant evaluates the score of the server in respect to each service from its list and causes the pheromone table, which contains scores for service-server placements, of the current server to be updated. The table is later on used by other ants to decide which server to visit next. When the ant has assigned all services, it reports its path to the service manager M_s and terminates. The service manager compares all reported paths using the partial objective function, which rates a specific placement, and decides about a rearrangement of the placement.

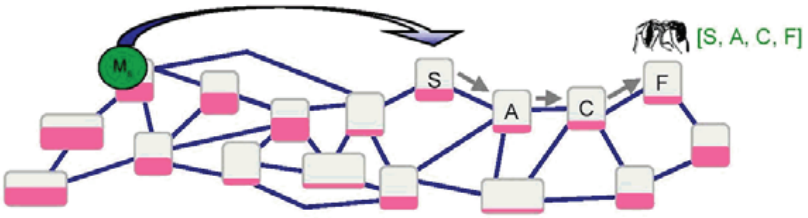


Figure 3.2: Placing of services by an ACO based algorithm (taken from [15])

Efficient Task Distribution using an Artificial Hormone System

In order to allocate tasks in a completely decentralized and self-organized way, Brinkschulte et al. proposed an algorithm, which is based on an artificial hormone system. The described strategy is intended for middleware based task allocation to heterogeneous processing elements, as illustrated in Figure 3.3.

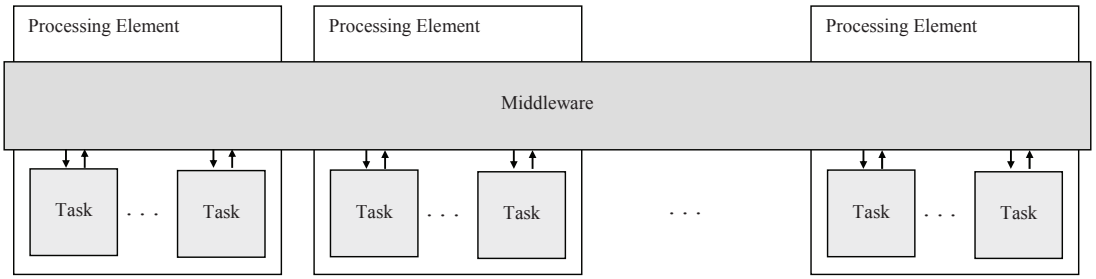


Figure 3.3: Processing Elements, Middleware and Tasks in the Artificial Hormone System Algorithm (taken from [7])

For the allocation of tasks to processing elements, three different types of hormones are used [7]:

- **Eager Value:** The Eager Value hormone determines how well a processing element can execute a task. The higher the Eager Value is, the better a processing element can execute a task.

- **Suppressor:** The Suppressor hormone suppresses the execution of a task on a processing element. Its values are subtracted from the Eager Values in order to prevent duplicate task allocation or to indicate a deteriorating state of a processing element.

- **Accelerator:** Values for the Accelerator favour the execution of a task on a processing element. They are added to the Eager Value and can be used to cluster cooperating tasks in the neighbourhood or to indicate an improved state of a processing element.

The basic control loop of the system (Figure 3.4), which is executed for every task on every processing element, works as follows:

- Determination based on the level of the three hormone types, if a task T_i is executed on a processing Element PE_γ or not.

- Local static Eager Value $E_{i\gamma}$ indicates how well the task T_i executes on PE_γ

- From this value, all Suppressors $S^{i\gamma}$ received for the task T_i on PE_γ are subtracted and all accelerators received for Task T_i on PE_γ are added resulting in a modified Eager Value $Em_{i\gamma}$ for task T_i on PE_γ.

- The modified Eager Value is sent by the middleware to all other processing elements of the system and compared to the modified Eager Values $Em^{i\gamma}$ received from all other processing elements for this task.

- Is $Em_{i\gamma}$ greater than all received Eager Values $Em^{i\gamma}$, the task T_i will be taken by PE_γ.

- Now task T_i on PE_γ sends Suppressors $S_{i\gamma}$ to all other processing elements to prevent a duplicate task allocation.

- Accelerators $A_{i\gamma}$ are sent to neighboured processing elements to favour the clustering of cooperating tasks.

- The described procedure is repeated periodically.

As each processing element is responsible for its own tasks, the described approach is completely decentralized. The communication to other processing elements is realized by a unified hormone concept, and the implementing system is thereby achieving several self-X properties: self-organizing, self-configuring, self-optimizing and self-healing [7]. Additionally, the algorithm is realtime capable, as there are tight upper time bounds for self-configuration [7].

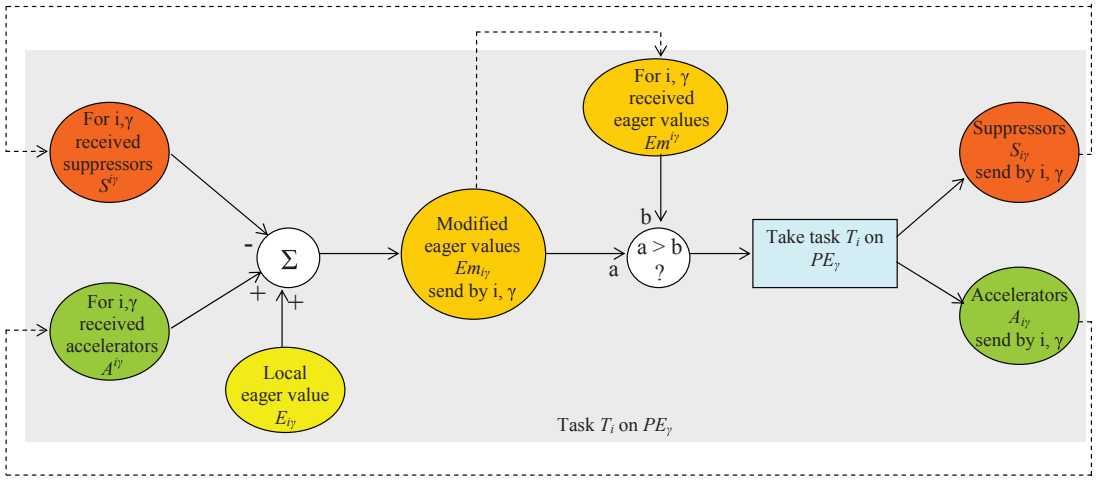


Figure 3.4: Control Loop for the Hormone Based Algorithm (taken from [7])

Self-Organization inspired by Cell Transformation and Locomotion

A framework, which offers a federation of components the possibility to adapt itself by using an approach originating from the transformation and locomotion behaviour of cells, has been proposed by Satoh [23]. The federation of components, which may run on heterogeneous computers, is thereby able to react to changes in user requirements and their associated context, such as user location and user tasks [23]. Components, which constitute the overall system, are implemented as mobile agents, that can travel from computer to computer under their own control. Additionally to these autonomous moving and duplication characteristics, one or more components can be combined as a virtual computer over distributed systems.

This combination or federation of components can be transformed and made mobile through bio-inspired self-organization, such as that undertaken by cells in their transforming and crawling locomotion. The proposed framework permits a component to speculatively deploy its clones at multiple neighbouring computers and to select one of the most appropriate clones. This behaviour corresponds to the process, lamellipodia bacteria go through in motile cells. Each component can have its own deployment policy for specifying spatial constraints between its location and the locations of other components at neighbouring computers.

Examples for possible deployment policies include (Figure 3.5):

- **Follow:** When a component declares “follow” to another component, then if the other component moves, the declarer or its clone migrates to the destination or a nearby proper host. This behaviour can also be defined as aggregation.

- **Dispatch:** This policy enables a component to stay in the current location and, upon the migration of a related component, a clone of the component is created and deployed at the new location of the related component.
- **Fill:** When a component declares "fill" for another component, then if the other component moves, the declarer or its clone migrates to the source of the latter component, or a nearby host. This behaviour can also be described as "tracking of footprints".
- **Shift:** The "shift" policy enables a component to simply follow the movement of another component.

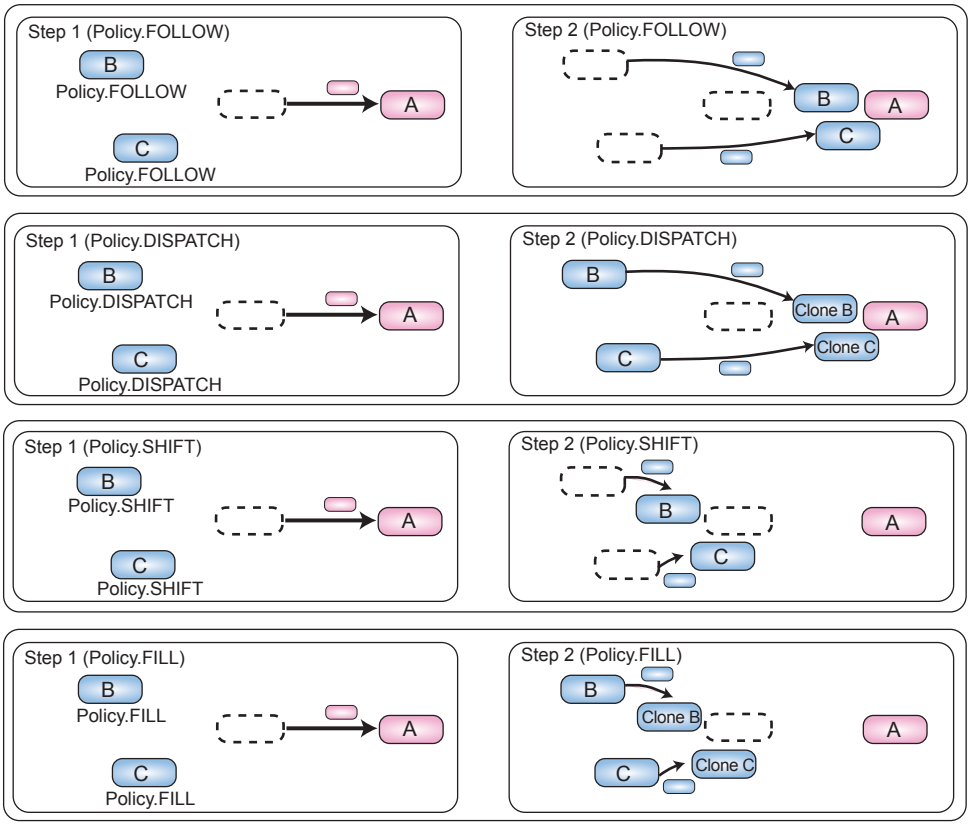


Figure 3.5: Migration policies in a Cell (taken from [23])

By enabling each component to migrate over a distributed system under its own policy, the federation as a whole is mobile and able to transform in a self-organized manner. In doing so, the system is able to adapt itself to changes in processing requirements in a completely decentralized way and without the need of any additional knowledge of the system environment.

Physically Inspired Algorithms: The Particles Approach

In contrast to the approaches described previously, physically inspired algorithms draw their inspiration out of phenomena observable in the world of physics. One of these algorithms, which has been proposed by [29], serves as a basis for the adaptation strategy developed in this thesis. Heiss and Schmitz have analyzed the problem of mapping tasks to processor nodes at run-time in multi-programmed multi-computer systems, and came up with an algorithm that uses the concept of physical forces to balance the system load, also known as *Particles Approach* [29].

Every task running inside the system is considered a particle on which several forces act upon, and each aspect of the allocation goal is modelled by a dedicated force. The process of load balancing can be thought of as a container in which several fluids of different viscosity reside. Upon the addition of a different fluid, representing the load increase, the fluids reorganize themselves according to the acting physical forces.

As an illustrating example, one could imagine a flat container with an even bottom and different amounts of non-mixable fluids of different viscosity, placed at different points (Figure 3.6). Gravitation forces the fluids to run, but frictional resistance and cohesion forces, that make up the viscosity, are working counter. Thinner fluids may spread out evenly across the bottom of the container, while more viscous fluids stick together like a lump. After adding an additional fluid, the distribution will reach a stable state with balanced forces (Figure 3.7) in a certain amount of time.

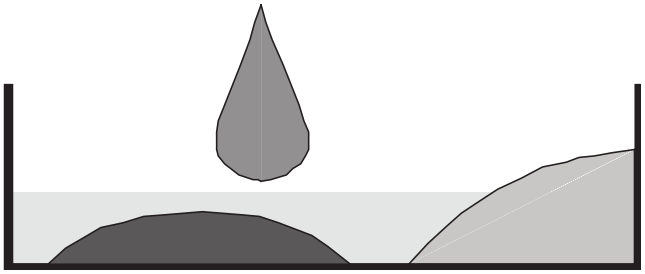


Figure 3.6: Adding a Fluid (taken from [29])

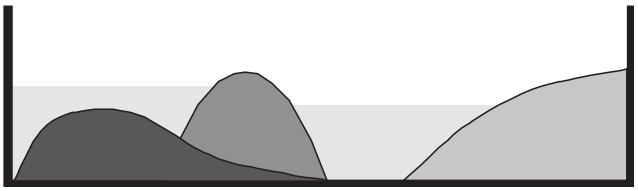


Figure 3.7: Equilibrium after adding fluid (taken from [29])

Using the image described before, the algorithm considers parallel computation as fluids with the tasks as particles. The load potential at each node in the system is used to define a gravitational force, whereas the communication relations along with their intensities are associated with a cohesion force in direction and magnitude. Finally, the costs to migrate a task are acting as a frictional resistance, which counteracts the load balancing force.

Thus, the algorithm pursues the following goals [29]:

- Minimization of load unbalances
- Minimization of communication costs
- Avoidance of unproductive migrations
- Stability, e.g. avoidance of oscillations

Parameters and Functions

In order to calculate the necessary forces acting upon the tasks, the parameters shown in Table 3.1 must be known and are used in the following chapters to define each force.

$a(k,l)$	Time it takes to transport one data unit from processor k to processor l
T	Set of tasks
$CsubseteqTxT$	Set of communication channels
$c(i,j)$	Set of communication channels
$loc(j)$	Length of task t_i (number of instructions to be executed)
s_i	Length of task t_i (number of instructions to be executed)
d_i	Size of description of task t_i (amount of data to be migrated)
r	Random variable drawn from a uniform distribution over the interval $[0,1)$
z_i	Number of performed migrations for a task

Table 3.1: Parameters for force calculation

Load Balancing Force

The Load Balancing Force between two adjacent nodes is defined as the ratio of their load potentials, and should counteract a node overload. To determine the load potential of a node, three different definitions are possible. The load potential of a node can be defined as the number of tasks assigned, as shown in Equation 3.1, the amount of work assigned, as shown in Equation 3.2, or it can be defined as the time it takes to execute the work, as shown in Equation 3.3.

$$V_{load}^k := |\{j : loc(j) = k\}| \tag{3.1}$$

$$V_{load}^k := \sum_{j:loc(j)=k} s_j \tag{3.2}$$

$$V_{load}^{k} := \frac{1}{\mu_k} \sum_{j:loc(j)=k} s_j \tag{3.3}$$

Defining the load potential of a node by the time it takes to execute the assigned work (Equation 3.3) is the most accurate strategy, as it considers different processor speeds and different sizes of the task. Finally, the load balancing force is defined as the ratio of the load potentials, as illustrated in Equation 3.4.

$$f_{lb}^{j \to k}(t_i) := c_{lb} \left(\frac{v_{load}^{j} + 1}{v_{load}^{k} + 1} \right) \tag{3.4}$$

Communication Force

The Communication Force is based on the communication intensities between the tasks, and should group the respective tasks together on the same node to minimize network transport. In order to calculate the Communication Force, the current position of each communication partner is needed. Additionally to that, the communication costs of a pair of communicating tasks must be known, which can be defined as the product of the amount of transferred data and the distance between the two tasks. The communication potential of a task t_i residing at node k is defined as its total communication costs, as illustrated in Equation 3.5.

$$v_{com}^{k}(t_i) := \sum_{j=1}^{|T|} a(k, loc(j))\ c(i, j) \tag{3.5}$$

Damping Force

The shipment and the installation of a task at the target node means cost, thus migration is only useful in cases, where the gain achieved by the migration outweighs the incurred cost. Therefore, the Damping Force can be seen analogous to the physical friction of a body, which acts as a counterforce to any other force attracting the body. Thus, any attracting force must exceed the friction in magnitude to move the body. Consequently, the Damping Force is defined as the negated value of its size multiplied by the distance between the two nodes, as shown in Equation 3.6.

$$f_{frict}^{j \to k}(t_i) := -c_{frict} d_i a(k, k) \tag{3.6}$$

In order to prevent processes from oscillating between two nodes, a second damping component is introduced that counteracts migration. Therefore, a migration counter z_i is used, which increases with each migration, and a constant defining the maximum amount of permitted migrations. Then, a migration is then only if the quotient of the current number of migrations, and the amount of migrations permitted, is smaller or equal to a random variable drawn from a uniform distribution over the interval $[0, 1)$, as illustrated in Equation 3.7.

$$x_i := \begin{cases} 1, & \text{if } r \geq \frac{z_i}{max_{migs}} \\ 0, & \text{otherwise} \end{cases} \tag{3.7}$$

Algorithm

In order to apply the presented concepts, it is necessary to define when the system should calculate the forces and migrate tasks, so when to execute a so called load balancing step. Load balancing should naturally take place when the load situation changed, thus, when tasks are generated or finished. Generating new tasks makes the generating site a possible sender of load, it therefore initiates a load balancing step. If tasks are finished, the finishing site informs all direct neighbours about the load change and continues normal operation. If necessary, the informed neighbours start the load balancing procedure. After collecting all required load information from its neighbours, the load balancing node determines, which of its tasks are eligible for migration by evaluating x_i according to Equation 3.7. For all eligible tasks, all forces are calculated that attract the task into one of the possible directions defined by the direct links according to Equation 3.8.

$$f_{res}^{j \to k}(t_i) := f_{lb}^{j \to k}(t_i) + f_{com}^{j \to k}(t_i) + f_{frict}^{j \to k}(t_i) \tag{3.8}$$

The maximum resulting force is taken as the total force acting on a task. Of all tasks that are attracted to a neighbour, we choose the one with the biggest resulting force and initiate its migration. After each migration, the source node informs its neighbours about the new load situation, which may lead to a domino effect of load balancing activities. At each node, load balancing stops if either no task is eligible for migration or all forces are negative.

3.2 Conclusion

The finding of algorithms to challenge the adaptation problem in large distributed systems in a decentralized manner is still an ongoing topic in current research. Some approaches have been presented that solve the problem at hand in many different ways. Each presented approach uses nature as innovative force and transfers beneficial characteristics of nature phenomena into their strategies. In the following chapters, one of these described algorithms, the Particles Approach, serves as a basis for an own algorithm. The Particles Approach is modified were necessary, implemented into the stream management system ESC and benchmarked at the end.

CHAPTER 4

Architecture of ESC

ESC is a cloud based stream processing engine designed for computations with real time demands following the actors pattern [3]. The system offers a simple programming model in which programs are specified by directed acyclic graphs (DAGs). The DAG defines the data flow of a program. Its vertices represent operations applied to the incoming data tuples [24]. Each tuple is composed out of the name of the processing element, a worker key, an event key and an event value. An example of an ESC setup with 4 slave nodes is illustrated in Figure 4.1.

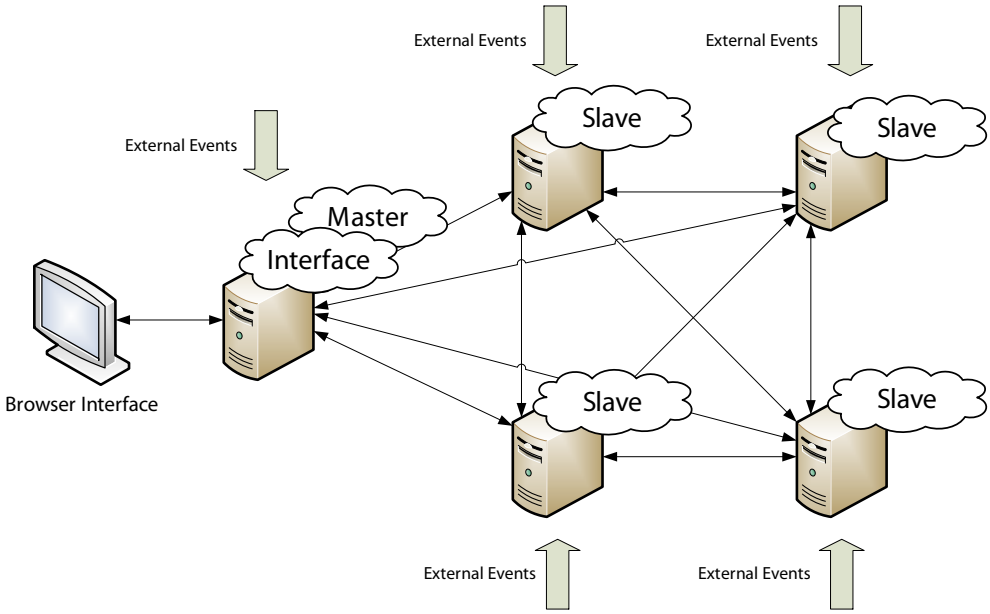


Figure 4.1: Illustration of an ESC setup with 4 slave nodes

4.1 Main Components of ESC

ESC is separated into three independent applications, which include the master application, the slave application and the interface application. Each application can run on a separate machine. Nevertheless, it is recommended to run the master application and the interface application together on the same node, and every slave application instance on a separate machine.

The main duties of the master application include the management of slave nodes, the initial placement of workers on nodes, as well as the start of scenarios. In contrast to that, the interface application is able to interact with a browser using the WebSocket Protocol [13], making it possible to monitor ESC within a browser window. Finally, the slave instances process received events according to the logic defined in modules assigned to the processing element, the received event belongs to. Furthermore, the slave instances distribute their load information, as well as their latency to every other slave instance within the network of nodes, to every registered node. Lastly, each slave instance is responsible for the creation and moving of worker instances.

4.2 Scenario Definition and Event Processing

Scenarios are defined using configuration files with an erlang-oriented syntax. Each configuration file contains a definition of the scenario graph, a section for assigning options to the included processing elements, and an area containing global scenario properties, like the applied load balancing strategy or the initial worker placement strategy. An example, which is a small version of the linear road scenario configuration file, is given in Listing 1.

```
{graph, [ % scenario graph as a list of adjacent PE-Ids
    {xway0, [car]}, {car, [accident_evaluator]},
    {accident_evaluator, []}
]}.

{pe_ids, [ % PE-Ids with module, arguments and options
    {xway0, [
        {module, lr_xway}, {arguments, []},
        {events, any}, {node, any} {hibernate, off},
        {movable, false}
    ]}
]}.

{properties, [ % global scenario properties
    {load_balancing_strategy, lb_none},
    {placement_strategy, pl_least_loaded}
]}.
```

Listing 1: Example Scenario Configuration File

Within the first section of the scenario configuration file, a graph is created by combining adjacent processing elements. Afterwards, in the second section, every specified processing element is configured. An overview of every configuration parameter, together with their possible assignments and a corresponding description, is given in Table 4.1.

Parameter	Value Range	Description
module	Module	The name of the module, which contains the worker implementation.
arguments	Arguments	A list of arguments for the worker instantiation. The default assignment is the empty list [].
events	[any | Events]	By defining a list of event keys, incoming events can be filtered. If all incoming events should be forwarded to the processing element, the assignment must be any.
node	[any | Node]	Definition of the node, where the processing element is created. The default assignment is any.
hibernate	[off | on]	If set to on, every worker instance will stay in a hibernate state, where it consumes much less memory. Upon reception of a new event, the worker instance gets activated, and returns to the hibernate state after the event has been processed. The default assignment is off.
movable	[$true$ | $false$]	If set to $false$, every worker instance of this PE stays on the node where it has been created. The default assignment is $true$.

Table 4.1: Options for Processing Elements

After a mapping of processing elements to corresponding modules within the scenario configuration file has been performed, the modules must be implemented. Therefore, similar to the implementation of an interface, three methods must be provided. To illustrate a minimal worker example, an implementation is given in Listing 2. Upon creation, termination, as well as upon reception of a new event, the designated method is called.

```
% initialize worker callback
initialize(WorkerState) -> WorkerState.

% terminate worker callback
terminate(WorkerState) -> ok.

% event processing
event(WorkerKey, {EventKey, EventValue}, WorkerState) -> ok.
```

Listing 2: Minimal Worker Example

4.3 Worker Migration

Migrating workers between nodes is crucial for achieving a balanced work distribution. In order to move a worker, two things have to be ensured. First of all, the process of moving must not slow down the processing of events designated to the worker, that should be moved. Second of all, the state of the worker after the move should be identical to the state, the worker would have had if no movement had taken place.

An approach designed to ensure the mentioned requirements is illustrated in Figure 4.2. In order for the worker to continue its work, a separate process is created which transfers the state of the worker to the new node. That way, no delay is introduced by the data transfer. Finally, in order for the copied worker to have the same state after the move, the original worker buffers every incoming event. Once, after the transfer of the worker state is complete, the buffered event queue is transferred and processed by the new worker, such that the state of the worker does not change by the worker move.

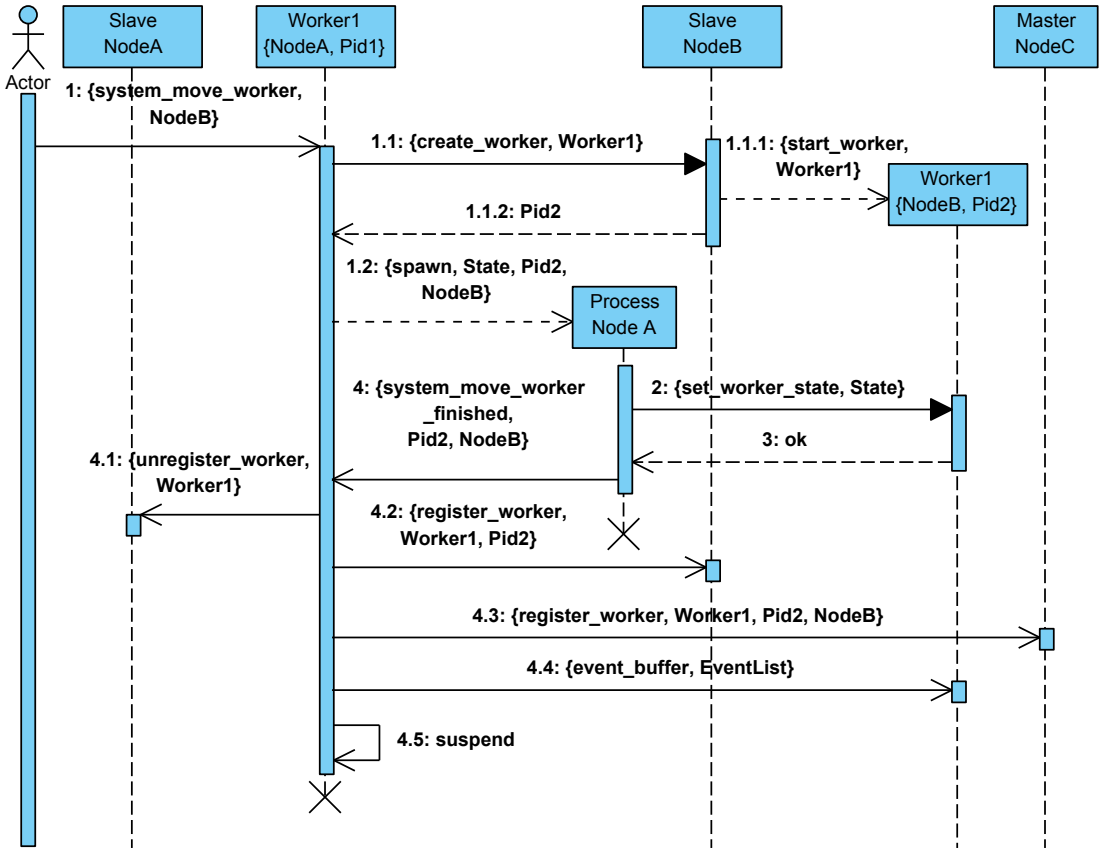


Figure 4.2: Sequence Diagram of a Worker Move

CHAPTER 5

Operator Placement Approaches

5.1 Initial Worker Placement

In order to create a new worker, a decision has to be made on where the new worker is to be placed. This decision is important, as an optimal placement strategy for a scenario reduces the need for costly worker moves and thereby improves overall performance. The optimal placement approach is only possible, if perfect knowledge exists about the amount and type of incoming tuples for every point in the future. Under these conditions, at any new worker creation the ideal node candidate is already known. But as these conditions do not apply for stream processing systems, where incoming data is generally unpredictable, only approximations to an optimal placement are possible.

Within this thesis, three different placement strategies have been developed and compared with each other. The first, and most simple, placement strategy randomly chooses a node from the list of available nodes. A second approach chooses the currently least loaded node as the new worker location. The last approach, *Similar ID*, tries to include knowledge about the current scenario in order to find an ideal node candidate.

Random Node

The first developed approach on how to assign workers to nodes is by picking a candidate randomly out of the available node list, as shown in Listing 3.

```
node_for_worker(_Worker = {_PeId, _WorkerKey}) ->
    Nodes = m_monitor:registered_nodes(),
    random:seed(now()), % initialize random generator
    lists:nth(random:uniform(length(Nodes)), Nodes).
```

Listing 3: Random Placement

Even though choosing a node randomly is a rather simple approach, it has two significant advantages. First of all, the decision on where to create the worker is very fast in contrast to other

strategies. Furthermore, the distribution of workers to nodes is uniformly distributed, which is especially an advantage when every worker has the same memory footprint.

Least Loaded Node

Another possibility is to create the worker on the currently least loaded node within the node network, as illustrated in Listing 4.

```
node_for_worker(_Worker = {_PeId, _WorkerKey}) ->
    m_monitor:get_least_loaded_node().
```

Listing 4: Least Loaded Placement

In order to know, where the currently least loaded node is located, a global load monitoring functionality has to be available. Within the analyzed application, every node sends its load to every other node in the node network in random intervals. Hence, the distribution of load information is completely decentralized.

Although the placing of a worker on the least loaded node reliefs every busy node and only assigns workers to nodes, which apparently have enough resources, this approach has two major disadvantages. Firstly, it is not possible to have absolute knowledge about the exact load of every node in the node network. That means, if a decision is made on where a node is to be placed depending on the node load, the load situation of this node could already have changed. Secondly, in situations where a lot of workers have to be created in a short amount of time, the least loaded node is flooded with new workers, leading to a significant imbalance of load distribution.

Similar Worker ID

In contrast to the strategies described previously, *Similar Worker ID* requires knowledge of the executed scenario, which in this case is Linear Road [4]. The developed approach, which is shown in Listing 5, assumes for example, that every worker key begins with the expressway its worker is assigned to.

```
node_for_worker(Worker = {_, WorkerKey}) ->
    case xway(WorkerKey) of
        u -> m_log:warn("Placement failed.");
        X -> node_for_xway(X)
    end.

% first element of the worker key is always the expressway
xway(Key) when (is_tuple(Key) and size(Key) > 0) ->
    element(1, Key);
xway(_) -> u. % undefined

node_for_xway(Expressway) ->
    Nodes = lists:sort(m_monitor:registered_nodes()),
    Pos = (Expressway rem erlang:length(Nodes)) + 1,
    Node = lists:nth(Pos, Nodes), Node.
```

Listing 5: Similar ID Placement

After the expressway is extracted from the worker key, a node is returned according to the given expressway, such that every expressway is always mapped to the same node. This behaviour is based on two predictions about the scenario itself and on the execution setup. First of all, an assumption is made that the amount of expressways corresponds to the amount of executing nodes. Secondly, the amount of load, that emanates from every expressway, is presumed to be more or less equal. If these predictions hold true, a balanced allocation of workers to nodes is guaranteed.

In order for this approach to work efficiently, good knowledge of the scenario setup, as well as of the incoming data, is required. Hence, the quality of this approach heavily relies on the made assumptions and their conformance with the observed behaviour during the scenario execution.

5.2 Load Balancing

The definition of an initial placement strategy for workers provides a way to counteract overload situations before they occur. But in situations, where the type and amount of incoming data, as well as the underlying scenario setup, are unknown or not predictable, additional mechanisms are needed to prevent a system overload and to guarantee the lowest possible latency. In order to achieve this, a load balancing strategy must be applied, that is moving worker between nodes when necessary and thereby achieving a potentially optimized worker distribution with improved quality of service characteristics.

Three different load balancing approaches have been analyzed within this thesis, upon which one of these approaches means no load balancing at all. Executing the scenario without a load balancing strategy should give an impression on whether or not an improvement is observable with the utilization of load balancing, or whether the additionally introduced computational overhead is causing a decrease in system performance.

Naive Approach

As the name already suggests, the *Naive Approach* does not include a sophisticated strategy on how to react if a load imbalance occurs. In fact, upon detection of a load imbalance, its sole measure to counteract is to move a random worker from the monitored node to the currently least loaded node within the node network, as shown in Listing 6.

```
handle_info(timeout, State) ->
    TargetNode = get_least_loaded_node(),
    balance_load(TargetNode),
    erlang:send_after(?TIMEOUT(), self(), timeout),
    {noreply, State};

balance_load(Node) ->
    Pids = s_monitor:registered_worker_pids(),
    move_random_pid(Pids, Node);

move_random_pid(Pids, TargetNode) ->
    P = lists:nth(random:uniform(length(Pids)), Pids).
    i_main:move_worker(P, TargetNode).

get_least_loaded_node() ->
    C = m_monitor:registered_nodes(), % node candidates
    S = lists:sort(fun(N1, N2) -> load(N1) < load(N2) end, C),
    N = lists:nth(1, S),
    case load(N) + ?DELTA < load(node()) of true -> N -> node().

load(N) -> m_monitor:get_node_load(N).
```

Listing 6: Naive Load Balancing

After a new scenario is loaded, a new server instance is created on every slave node, which is responsible for the load balancing. The created server instance than analyzes the load situation

periodically, leaving a random time interval between each check to prevent the initiation of a load balancing step by multiple nodes at once. When a slave instance is executing a load balancing step, the load information received from every other slave node is compared with the own load and a decision is made, whether the need for load balancing exists, or not. If so, a random worker is picked from the list of workers on the slave node and placed on the currently least loaded node within the node network.

The Particles Approach

In contrast to the previously described strategy, the *Particles Approach* takes significantly more factors into account before making load balancing decisions. The approach is adapted from the algorithm described in Chapter 3.1, where it was presented in combination with the problem of mapping tasks to processor nodes at runtime. Despite the fact that the original algorithm was designed to work in a different environment, the transition of major parts was possible without any significant modification. The main section of the algorithm is given as a simplified extract in Listing 7.

```
incoming_event(_, {SenderPeId, _}, SenderPos, _, State) ->
    CI = communication_intensities(SenderPeId, SenderPos),
    TimeSinceLastUpdate = timer:now_diff(now(), last_update()),
    handle_event(self(), CI, TimeSinceLastUpdate, State).

handle_event(Pid, CI, T, State) when T > ?UPDATE_FORCE_INTERVAL ->
    {BestNode, Force} = get_best_node(CI),
    MoveCounter = State#slave_worker.move_counter
    update_force(BestNode, Force, MoveCounter, self()),
    erlang:put(com_int, undefined); % clear communications dictionary
handle_event(_, CI, _, _) -> put(com_int, CI).

get_best_node(CI) ->
    lists:foldl(fun(N, {MaxN, MaxF}) ->
            F = lb_force(N) + comm_force(N) + damping_force(N),
            case F > MaxF of true -> {N, F}; false -> {MaxN, MaxF} end
        end, {node(), 0}, m_monitor:registered_nodes()).

update_force(N, F, MC, Pid) -> gen_server:cast(?M, {move, N, F, MC, Pid}).
```

Listing 7: The Particles Approach - Implementation

Just like in the *Naive Approach* described before, upon initialization of a scenario, a new server instance is started on every slave node. As well as in the *Naive Approach*, the server instance initiates a load balancing step after a randomly generated timeout. But, in contrast to the previous approach, the worker that is drawn to another node with the highest force is moved, whether there exists a load imbalance, or not. After a worker has been moved, the list of workers and their attracting forces is cleared.

In order for the server instance to get hold of the attracting forces, each worker is responsible for the determination of the node, it is drawn to the most, and the forwarding of this information to the server instance. As the force calculation is computationally expensive, it is only performed

after a minimum time interval, which is checked for each worker separately. If a new event is received within the minimum time interval, only the communication intensity with the sending node is increased and saved in the process dictionary. Should the new event occur after the minimum time interval, the attracting force to every node within the node network is calculated. If the node with the highest attracting force is not the current node, a message is sent to the server instance, containing the process id of the worker, the total force and the attracting node. As described before, the server instance then moves the worker with the highest attracting force.

Load Balancing Force

As the load of every node is distributed periodically via multicast, every worker has instant access to calculate the load balancing force, as shown in Listing 8.

```
% load balancing directed to a target node
lb_force(Node) -> trunc(?WEIGHT * f(l(node()), l(Node))),

f(MyLoad, TargetLoad) when MyLoad < TargetLoad - ?DELTA ->
    (-1) * ((TargetLoad + 1) / (MyLoad + 1));
f(MyLoad, TargetLoad) when MyLoad > TargetLoad + ?DELTA ->
    (MyLoad + 1) / (TargetLoad + 1);
f(_, _, _, _) ->
    0. % return 0 if load is in DELTA interval

l(N) -> m_monitor:get_node_load(N).
```

Listing 8: Load Balancing Force

In contrast to the original algorithm, the load balancing force can become negative and depends on a defined delta value. On the one hand, if the load of the examined node is higher than the load of the original node, plus a defined delta value, a positive force is returned. On the other hand, when the load of the examined node is smaller than the load of the original node, minus a defined delta value, the returned force is negative. If none of these conditions holds true, the load balancing force is zero.

Communication Force

For the determination of the communication force, a node is imposing on another node, the difference between the communication potential of the original node and the communication potential of the original node is calculated and multiplied with a defined weight constant, as illustrated in Listing 9.

The communication potential of a node for a worker is determined by multiplying the intensity of every communication with the latency between the given node and the communication partner node. Consequently, the result of subtracting the two communication potentials yields a statement about the possible improvement in communication latency, that could be gained by moving the worker to the analyzed node.

```
% communication force directed to a target node
comm_force(Node) ->
    trunc(?W * (com_pot(node()) - com_pot(Node))).

com_pot(Node) -> % communication potential
    CI = communication_intensities(),
    dict:fold(fun({_, N}, I, S) -> S + latency(Node, N) * I end, 0, CI).

latency(N1, N2, _) when N1 =:= N2 -> 0;
latency(N1, N2) ->
    case lists:keyfind({N1, N2}, 1, s_monitor:get_latencies()) of
        {_, L} when L < 1 -> 1; {_, L} -> L; _ -> 1
    end.
```

Listing 9: Communication Force

Damping Forces

Every moving of a worker from one node to another node is expensive, as it means an additional transfer of data within the node network. Hence, a force is required that prevents unprofitable worker migrations, as well as oscillating worker moves, where the best suitable node for a worker changes periodically in short time slots. The first damping force component, presented in Listing 10, leads to the preferred moving of workers that have, on the one hand, a low memory footprint, and, on the other hand, a small latency to the desired target node.

```
% damping force directed to a target node
damping_force(Node) ->
    {memory, Memory} = process_info(self(), memory),
    trunc(?W * (-1) * s_monitor:get_latency(Node) * Memory).
```

Listing 10: Damping Force 1

Inherent to every worker is a move counter, that is increased on every worker move. As shown in Listing 11, the current move counter is used, in conjunction with a constant defining the amount of maximum migrations, to prevent workers from moving periodically between nodes. A situation, where this kind of worker oscillation occurs, is when multiple attracting forces of approximately the same amount exist for a worker and the most suitable node candidate is changing frequently.

```
% decision, whether a new move candidate is accepted, or not
move(W, Pid, F, Node, MC) ->
    case random:uniform() >= (MC / ?MAX_MIGRATIONS) of
            true -> new_worker_entry(Worder, Pid, F, Node);
            false -> null
    end.
```

Listing 11: Damping Force 2

CHAPTER 6

Benchmarking Distributed Stream Processing Systems

The operator placement approaches described previously, together with the system architecture presented in Chapter 4, form a stream computing system with mechanisms to adapt itself, on the one hand, according to the arriving data, and, on the other hand, under consideration of the load situation within the node network. But, without a statement regarding the performance of the developed approaches, no conclusion can be drawn if any gain in the quality of service characteristics has been achieved. Therefore, a benchmarking system is needed to provide the desired results, and to offer a comparison of different strategies with each other, as well as a comparison with other stream processing systems. The only benchmark, that is available for stream processing systems, and for which test results with other stream processing systems exist, is *Linear Road*, which is presented in the following chapters.

6.1 Benchmarking with Linear Road

The simulation of streaming data poses a unique challenge to the design of a benchmark. For instance, the input data must have semantic validity and cannot be random. As a typical data stream presents discrete measurements of a continuous activity, the content of the data stream should be consistent with this activity. Furthermore, the absence of a query language standard for stream queries means that the benchmark queries must be specified in a more general, though unambiguous way. Ultimately, performance metrics for a stream processing benchmark should be based on response time, rather than completion time [4].

As stream queries are predominantly continuous, the typical database benchmark metric of "completion time" is inappropriate, given that such queries never complete. A more appropriate metric for streams could be "response time", meaning the average or maximum difference between the time, that an input arrives, and the time, outputs are computed. Furthermore, a

possible metric could be “supported query load”, meaning how much input a stream system can process while still meeting specified response times and correctness constraints [4].

After the benchmark has been performed, its output must be verifiable, even though results returned may vary depending on when and how they have been generated. As the results of continuous queries may depend upon evolving historical state, or the arrival order of tuples, several different results for the same query may be “correct”. Hence, the validation should account for queries that have multiple correct answers [4].

The Linear Road benchmark has been designed to meet all of the described challenges. Linear Road simulates a toll system for a motor vehicle expressway of a large metropolitan area. The tolling system applies “variable tolling”, which means it uses dynamic factors, such as traffic congestion and accident proximity, to calculate toll charges. Further features of the Linear Road benchmark include accident detection and accident alert notifications, traffic congestion measurements, toll calculations and the answering of historical queries [4].

A system implementing the Linear Road benchmark must maintain statistics about the number of vehicles and the average speed on each segment of each expressway on a per minute basis. Furthermore, the system has to detect accidents, deliver accident alerts to nearby vehicles and calculate toll charges dynamically, based on segment statistics and proximate accidents, as well as notify cars of their charges and assess the calculated tolls. Each query answer must satisfy the response time and correctness requirements specified. Then, the throughput that a system can sustain while meeting these requirements, which is measured in the number of expressways L, constitutes the benchmark score (L-Rating) [4].

Linear City

Linear City is composed out of parallel expressways, that run horizontally ten miles apart, as illustrated in Figure 6.1. For simplicity, there are no expressways in vertical direction. Each expressway has four lanes in eastbound direction, and four lanes in westbound direction. Of these four lanes, three are travel lanes and one lane is devoted to entrance and exit.

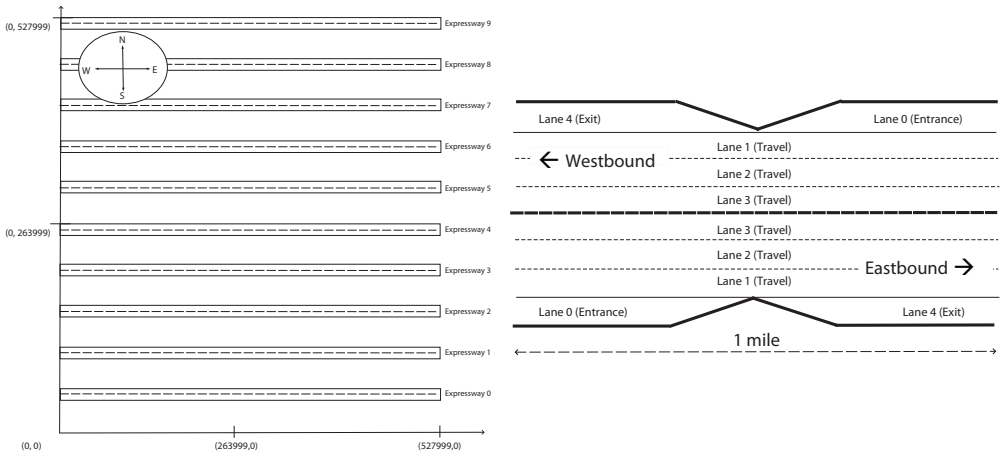


Figure 6.1: Geometry of Linear City (taken from [4])

Input Data

Two files are produced by the input data generator, which is shipped together with the Linear Road benchmark. The first file contains historical data, summarizing ten weeks of tolling history, that must be maintained by the system to answer historical query requests that refer to data dating prior to the start of the simulation. Hence, the data contains tuples for toll history of the form (VID, Day, $Xway$, $Tolls$), such that $Tolls$ is the total amount of tolls spent on expressway $Xway$ on day Day by vehicle VID. Before the simulation is started, the contents of the history file must already have been loaded in order to reply correctly to historical queries [4].

The second file contains the stream data, generated by the MIT Traffic Simulator [30]. Thus, the file consists of position reports of cars, account balance requests, daily expenditure requests and travel time requests. A position report is a tuple of the form $(Type = 0, Time, VID, Spd, XWay, Lane, Dir, Seg, Pos)$, where $Time$ is the time stamp identifying the time at which the position report was emitted, VID is an integer identifying the vehicle that emitted the position report, Spd is the speed, and $XWay$, $Lane$, Dir, Seg together with Pos indicate the vehicles position. In contrast to that, an account balance request is a tuple of the form $(Type = 2, Time, VID, QID)$, where $Time$ is the time when the request occurs, VID is the vehicle making the request and QID is an integer query identifier. Whenever historical data is queried, a daily expenditure request of the form $(Type = 3, Time, VID, XWay, QID, Day)$ is issued, such that VID is the vehicle issuing the request, QID is an integer query identifier, $XWay$ and Day identify the expressway and the day, where 1 stands for yesterday and 69 for 10 weeks ago. Finally, a travel time request is a tuple of the form $(Type = 4, Time, VID, XWay, QID, s\_init, s\_end, DOW, TOD)$, such that VID is the vehicle issuing the request, QID is the integer query identifier, $XWay$ is the expressway upon which the journey occurs (from segment s_init to segment s_end), DOW and TOD specify the day of the week and the minute, when the journey would take place [4].

Benchmark Requirements

In order for a system to pass the Linear Road benchmark, it needs to fulfill several requirements. First of all, the system needs to meet all requirements regarding toll notifications, which are summarized in Table 6.1. The trigger for the sending of a toll notification is an incoming position report. If the car, that sent the position report, has reached another segment, other than the segment it sent its last position report from, and the current lane is not an exit lane, a toll notification is issued and returned to the car, informing it about the assessed toll. The notification contains the last average speed within the last five minutes of the segment, that has been left by the car. Furthermore, the assessed toll is attached to the notification, calculated incorporating the last average speed, as stated before, and the existence of an accident in a segment up to four segments upstream. Lastly, the returned notification must be issued not more than five seconds after the position report has been sent [4].

Trigger	Position report, q
Preconditions	q . Seg $\neq \overleftarrow{q}$. Seg, $l \neq$ EXIT
Output	(Type: 0, VID: v, Time: t, Emit: t', Spd: $Lav(M(t), x, s, d)$, Toll: $Toll(M(t), x, s, d)$)
Recipient	v
Response	$t' - t \leq 5\,Sec$

Table 6.1: Toll Notification Requirements (taken from [4]))

The second type of requirement involves accident alerts, which is summarized in Table 6.1. As well as in the requirement before, the trigger for an accident alert is a position report. If a car changes its segment, the current lane is not an exit lane, and an accident exists within five segments downstream, an accident notification is issued and sent to the car. An accident on a position is present, whenever an identical position report is sent by more than one car more than three times in a row. In case that one of the cars issues a position report indicating that it moved from the position, where the accident occurred, and the remaining amount of cars is below one, the accident is assumed to be cleared. The accident alert has to be sent not later than five seconds after the triggering position report was issued [4].

Trigger	Position report, q
Preconditions	$\exists_{s', 0 \leq i \leq 4}(s' = Dn(q\,.\,\text{Seg}, d, i) \wedge$ $Acc\_in\_Seg(M(t) - 1, x, s', d))$, q . Seg $\neq \overleftarrow{q}$. Seq, $\neq$ EXIT
Output	(Type: 1, Time: t, Emit: t', Seg: s')
Recipient	v
Response	$t' - t \leq 5\,Sec$

Table 6.2: Accident Alert Requirements (taken from [4]))

A car can query its current toll balance and issue an account balance request. The required output, containing the current toll of the querying car, is described in Table 6.3. The system has to answer to an account balance query within a time interval of five seconds [4].

Trigger	Account balance request, a
Preconditions	$-$
Output	(Type: 2, VID: v, Time: t, Emit: t', ResultTime: τ, QID: q, Bal: $\sum_{p \in tollset(v)}, (f(p)))$ $s.t.$ p . Time $\leq \tau$, p . Seg $\neq Last_1(v,t)$. Seg) $f(p) =$ $Toll(M(p$. Time$), p$. XWay$, p$. Seg$, p$. Dir$)$
Recipient	v
Response	$t' - t \leq 5\,Sec$
Accuracy	$\tau \geq t - 60\,Sec$

Table 6.3: Account Balance Requirements (taken from [4]))

The last requirement type involves daily expenditure queries, requesting the toll a car has gathered for a given day. A response for this kind of query has to be issued within ten seconds, in order to pass the benchmark, as can be seen in Table 6.4.

Trigger	Daily Expenditure request, d
Preconditions	$-$
Output	(Type: 3, Time: t, Emit: t', QID: q, Bal: $\sum_{p \in tollset(v)}, (f(p)))$ $s.t.$ $Day(p$ $-$ Time$) = d$, p . XWay $= x$ $f(p) =$ $Toll(M(p$. Time$), p$. XWay$, p$. Seg$, p$. Dir$)$
Recipient	v
Response	$t' - t \leq 10\,Sec$

Table 6.4: Daily Expenditure Requirements (taken from [4]))

6.2 Linear Road Benchmark Implementation in ESC

An implementation for the Linear Road benchmark for the ESC stream processing system has been performed within the scope of this thesis. The developed scenario uses 12 processing elements and is written in Erlang. A graph, showing the connections between the processing elements, is illustrated in Figure 6.2.

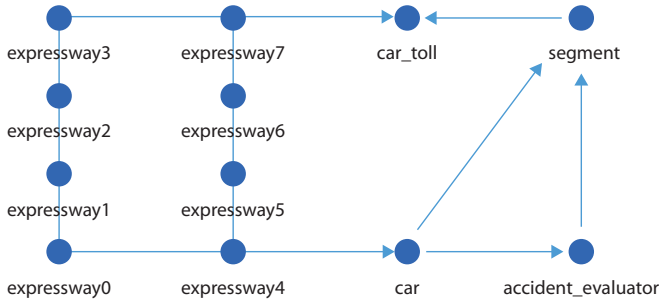


Figure 6.2: Linear Road Scenario DAG

For each processing element, worker instances are created and distributed between the slave nodes. Every worker instance has a worker key, allowing the work for a processing element to be distributed and therefore be processed in parallel. The flow of events, and the division of processing elements into worker instances, is illustrated in Figure 6.3.

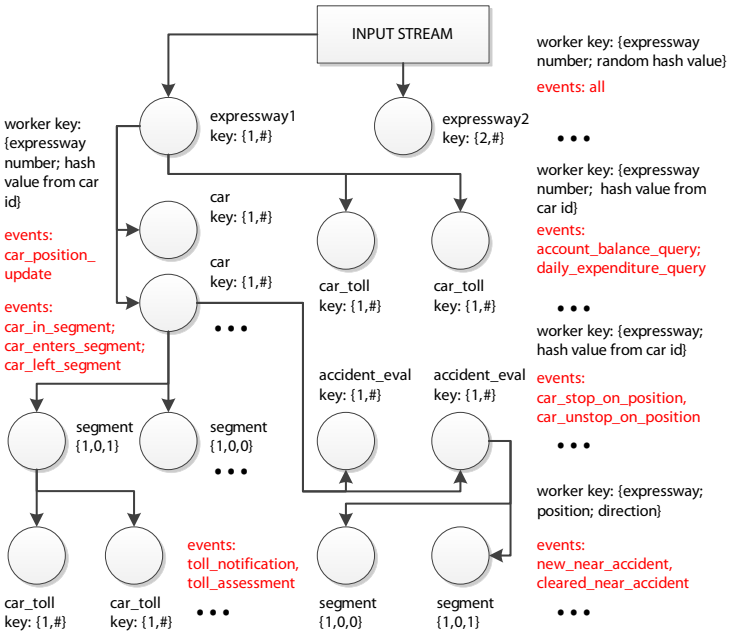


Figure 6.3: Linear Road Implementation

CHAPTER 7

Results

By the combination of the system illustrated in Chapter 4, together with the presented operator placement approaches in Chapter 5, and the execution of the benchmark described in Chapter 6, results concerning the effectiveness of the developed system and the adaptation approaches can be obtained. Therefore, the chapter is organized as follows:

- Presentation of the evaluation framework (Chapter 7.1)
- Description of the test results using the Linear Road benchmark (Chapter 7.2)
- Comparison of the results with other stream processing engines (Chapter 7.2)

7.1 Evaluation Framework

The evaluation framework used is predefined by the Linear Road benchmark described in Chapter 6.1. It is currently the only evaluation framework available for distributed stream processing systems, which has been tested with a wide range of other systems. Due to the wide usage, many test results are already available and can be used for comparison. The following chapters describe the used hardware scenarios and the input data characteristics.

Historical Data and Stream Datasets

The input data for the Linear Road benchmark is generated by the MIT Traffic Simulator [20]. For each expressway, the generation of stream data takes about 5 hours, depending on the speed of the generating computer. The data is stored in a data file which consists of position reports, account balance queries, daily expenditure queries and travel time estimation queries as comma separated values. For every expressway, a separate file is generated by the MIT Traffic Simulator and, after the data generation for every expressway has finished, the data is merged by the Linear Road data generator in order to prevent duplicate car ids and to have cars crossing expressways.

The generated data corresponds to three hours of traffic volume, where the quantity of generated events is rising each minute, as shown in Figure 7.1. Further on, the following characteristics are ensured:

- Each vehicle undertakes at least one journey during the simulation. A journey always begins on an entry ramp and finishes on an exit ramp.
- Every car emits a position report every 30 seconds and never travels faster than 100 mph, thus, it emits at least one event from every segment it travels on.
- It is guaranteed, that each vehicle has an average speed of 40 mph or less when entering or leaving an expressway, thus, emitting at least one position report from an entrance ramp and one position report from an exit ramp for every journey.
- An accident is created in a random location on each expressway for every 20 minutes of position reports and takes anywhere between 10 and 20 minutes to be cleared.
- With 1 % probability a report is accompanied with a historical request.
- About 50 % of the requests are account balance queries, 10 % are requests for daily expenditures and about 40 % are request for travel time estimation.
- The 3 hour simulation of a single expressway consists of 12 million position reports, about 67.000 account balance queries and 12.000 daily expenditure queries.
- The amount of records increases from 14 records per second at the start of the simulation to about 1700 records per second at the end.

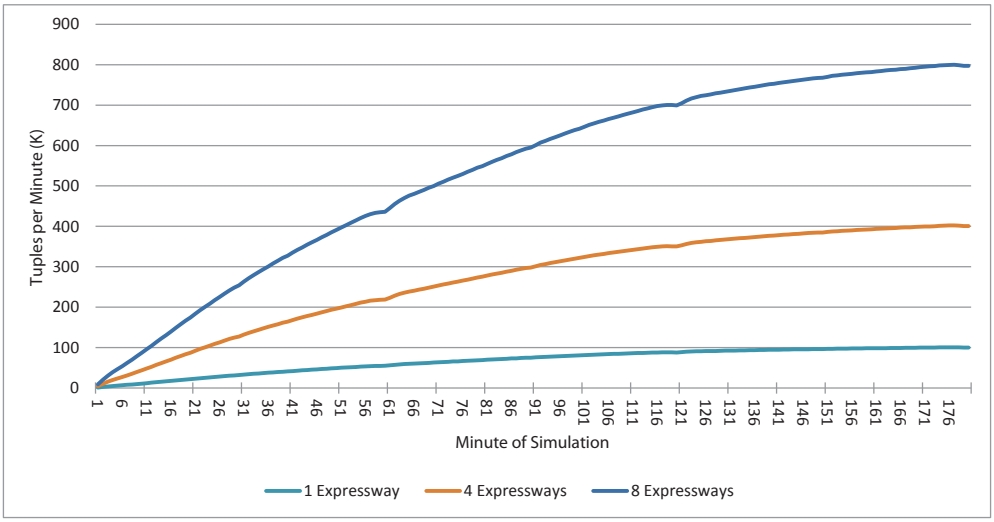


Figure 7.1: Load Distribution

Evaluation Measures

The most important measure for evaluating a distributed stream processing system with the Linear Road benchmark is the amount of expressways a system can process without exceeding the limits regarding the response time. Within the specification of the Linear Road benchmark, limits are defined for every type of query, which must not be exceeded by the executing system. The limits for each request type are summarized in Table 7.1.

Request Type	**Response Time Limit**
Toll Notification	$t' - t \leq 5\ sec$
Accident Notification	$t' - t \leq 5\ sec$
Account Balance Request	$t' - t \leq 5\ sec$
Daily Expenditure Request	$t' - t \leq 10\ sec$

Table 7.1: Response Time Requirements for Linear Road

In order to compare different configurations of ESC with each other, the number of expressways as evaluation measure is not enough. Instead, it is necessary to use the exact input for each of the different configurations to gain reliable results concerning their performance. Therefore, the response time for each query is taken as an evaluation measure. Finally, the average of all response times and the maximum value are used for rating purposes.

Test Scenario Environment

The Linear Road benchmark has been executed on three independent machines of the same hardware configuration. Each machine has been equipped with a 16 core CPU at a speed of 2.40 GHz, possessing 32 GB of memory. Furthermore, Debian Linux 6 has been used as operating system together with Erlang in version R15B02. For the execution of the master application, one of the three machines has been used. Hence, the slave instances have been distributed on the remaining two machines. In order for the slave instances to run on separate cores, a CPU topology has been defined [12], allocating a specific CPU core to the scheduler of the Erlang VM. A summary of the different environments is given in Table 7.2.

Characteristic	**Master Node**	**Slave Node**
CPU	16 Core CPU @ 2.40 GHz	1 Core CPU @ 2.40 GHz
Memory	32 GB RAM	6 GB RAM
Process Limit	100.000	100.000.000

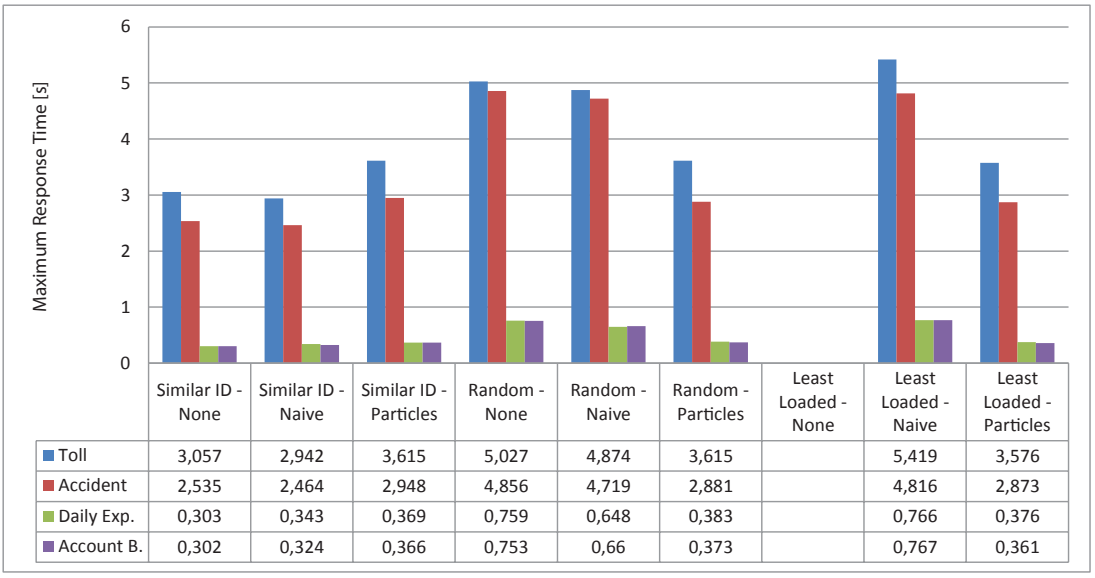
Table 7.2: Test Scenario Environment

7.2 Test Results

The Linear Road benchmark has been executed with 4 and 8 slave instances, as well as with 4 and 8 expressways, leading to eight different scenario environments. Within each environment, every combination of initial operator placement strategy and load balancing approach has been tested, resulting in 36 executions of the Linear Road benchmark with a total time of 108 hours.

As in previously published implementations of Linear Road [4] [16] [6], the travel time estimation was not implemented. Therefore, queries of this type were ignored and do not appear in the test results. In case no values appear in the chart for a specific configuration, the scenario did not complete because of a node overload. Nevertheless, a complete and more detailed overview of the maximum response times of every scenario is given in Appendix A.

4 Slaves - 4 Expressways



	Similar ID - None	Similar ID - Naive	Similar ID - Particles	Random - None	Random - Naive	Random - Particles	Least Loaded - None	Least Loaded - Naive	Least Loaded - Particles
Toll	3,057	2,942	3,615	5,027	4,874	3,615		5,419	3,576
Accident	2,535	2,464	2,948	4,856	4,719	2,881		4,816	2,873
Daily Exp.	0,303	0,343	0,369	0,759	0,648	0,383		0,766	0,376
Account B.	0,302	0,324	0,366	0,753	0,66	0,373		0,767	0,361

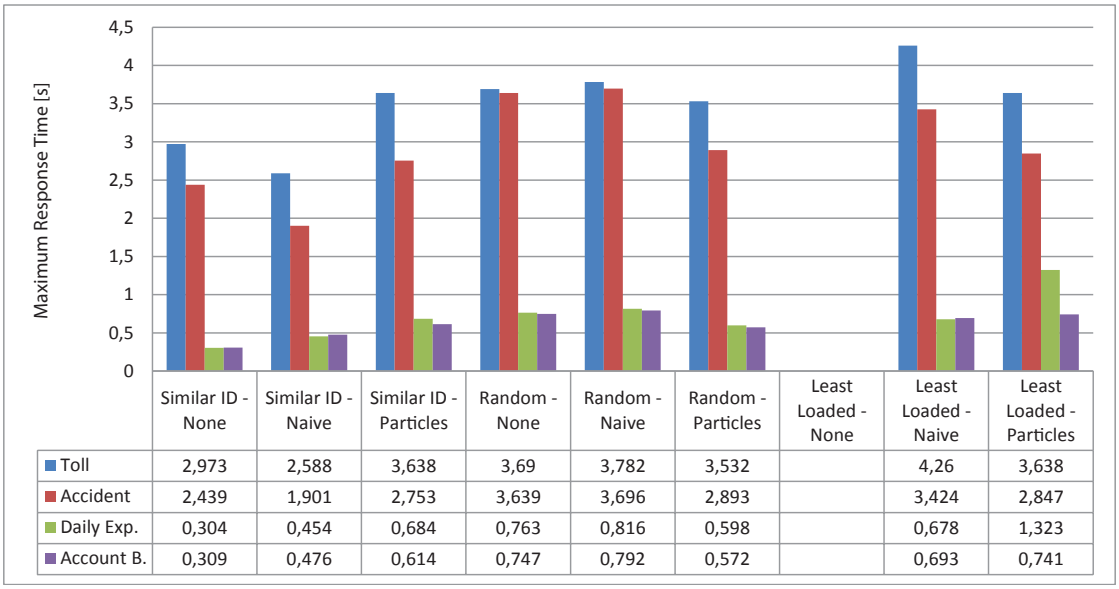
Figure 7.2: Test Results for 4 Slaves and 4 Expressways

As the chart in Figure 7.2 shows, nearly every scenario completed the benchmark in the environment *4 Slaves - 4 Expressways*, with the exception of the configuration *Least Loaded - None*. Of the configurations, that did complete the benchmark, the configurations *Random - None* and *Least Loaded - Naive* did not fulfill the requirements, as the maximum response time lies above 5 seconds. Consequently, the initial placement strategy, that performs best, is *Similar ID*, as the lowest response times have been measured, without regard to the used load balancing strategy. Furthermore, when comparing the load balancing strategies with each other, the *Particles* approach performed best, as constantly low maximum response times have been measured.

4 Slaves - 8 Expressways

None of the configurations did complete in the *4 Slaves - 8 Expressways* environment. Due to the huge amount of incoming events, every configuration came to a stop due to a node overload before the simulation has been completed. Hence, no statement can be made, whether or not the necessary requirements have been met, or which of the tested scenarios performed best. Nevertheless, a detailed overview about the maximum response times of each minute during the simulation is attached in Appendix A.

8 Slaves - 4 Expressways

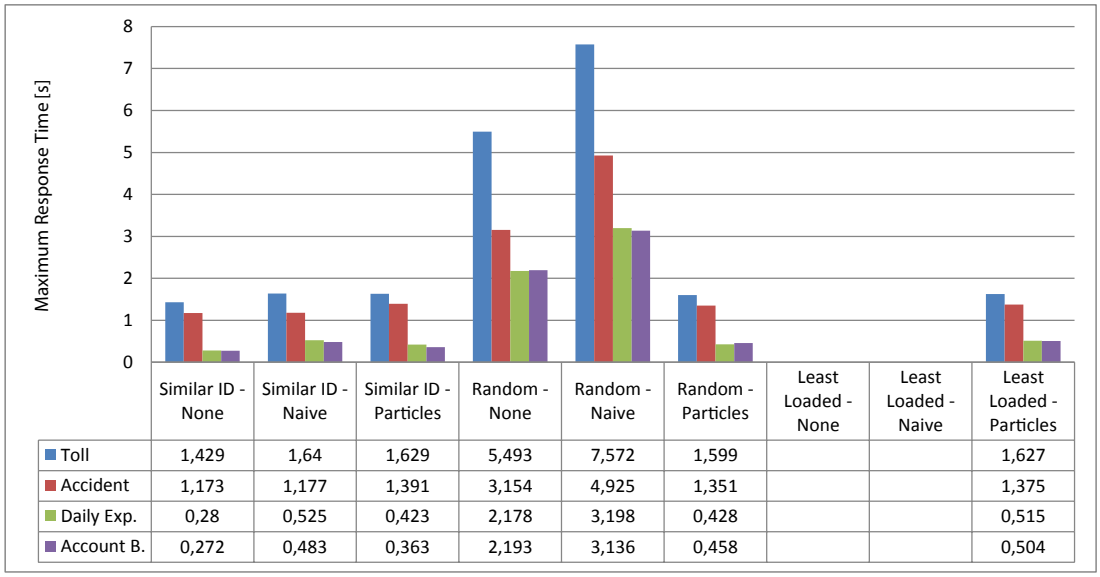


	Similar ID - None	Similar ID - Naive	Similar ID - Particles	Random - None	Random - Naive	Random - Particles	Least Loaded - None	Least Loaded - Naive	Least Loaded - Particles
Toll	2,973	2,588	3,638	3,69	3,782	3,532		4,26	3,638
Accident	2,439	1,901	2,753	3,639	3,696	2,893		3,424	2,847
Daily Exp.	0,304	0,454	0,684	0,763	0,816	0,598		0,678	1,323
Account B.	0,309	0,476	0,614	0,747	0,792	0,572		0,693	0,741

Figure 7.3: Test Results for 8 Slaves and 4 Expressways

Except the *Least Loaded - None* configuration, all scenarios did complete the benchmark and fulfilled the necessary requirements, as illustrated in Figure 7.3. As the imposed load is small in contrast to the amount of available slave instances, no huge difference in response time has been observed. Therefore, no optimal strategy could be identified within the analyzed environment.

8 Slaves - 8 Expressways



	Similar ID - None	Similar ID - Naive	Similar ID - Particles	Random - None	Random - Naive	Random - Particles	Least Loaded - None	Least Loaded - Naive	Least Loaded - Particles
Toll	1,429	1,64	1,629	5,493	7,572	1,599			1,627
Accident	1,173	1,177	1,391	3,154	4,925	1,351			1,375
Daily Exp.	0,28	0,525	0,423	2,178	3,198	0,428			0,515
Account B.	0,272	0,483	0,363	2,193	3,136	0,458			0,504

Figure 7.4: Test Results for 8 Slaves and 8 Expressways

Simulating 8 expressways implies the handling of up to 800.000 tuples per minute. Therefore, this configuration denoted the most challenging task for the analysed stream processing system. Two of the observed scenarios did not finish the simulation, whereas two of the remaining analyzed configurations did not meet the imposed requirements. Therefore, the best performance regarding the initial operator placement has been achieved by *Similar ID*, while the best performing load balancing approach within the analyzed environment turns out to be the *Particles* approach, which can be observed in Figure 7.4.

Comparison to other Stream Processing Engines

Results for other stream processing systems, executing the Linear Road benchmark, are presented in Table 7.3. By looking at these results it becomes obvious, that scalability is one of the major challenges still being faced. So it is possible to achieve a rating of 2.5 with one core, but as soon as the number of cores rises, the amount of expressways per core decreases significantly. A ratio of one expressway per core, when executing on more than one core, is a result, which must not fear the comparison with other stream processing engines, that have been benchmarked by Linear Road.

Name	**Year**	**L**	**#cores**	**Comment**
Aurora	2004	2.5	1	
SPC	2006	2.5	170	3 GHz Xeon
XQuery	2007	1.5	1	
scsq-lr	2007	1.5	1	Laptop
DataCell	2009	1	4	1.4 s average response time
stream schema	2010	5	4	
CaaaS	2011	1	2	Streaming MapReduce
ESC	2012	4	4	
ESC	2012	8	8	

Table 7.3: Linear Road Result Comparison (on the basis of [33])

CHAPTER 8

Conclusion and Future Work

In this thesis, adaptability approaches for distributed stream processing systems have been developed, and the performance of each strategy has been analyzed and compared with each other. Therefore, the main focus of this thesis has been the determination of an operator placement approach, which is best suited for the use in distributed stream processing systems.

It has been shown, that the developed initial placement strategy *Similar ID*, which assigns operators to nodes according to their unique key within the system, performs best. Furthermore, if an optimal placement of operators is not prevailing within the system, the *Particles* load balancing strategy turned out to move the operators in a way, that the system performed better than with any other examined load balancing approach.

Despite the low response times, that have been observed by the use of the *Similar ID* strategy, as well as by the use of the *Particles* load balancing, it should be noted, that none of these approaches is generally suited for every possible scenario. In order for both approaches to deliver optimal results, the scenario, that is being executed, must allow a best-suited operator distribution and the implementations for *Similar ID* and *Particles* need to guide the system in the direction of an ideal operator placement.

After the foundation for an advanced distributed stream processing system has been laid by the creation of this thesis, there are many possibilities for the extension of its functionality and the optimization of its performance. Since it is still possible to overload the system, and thereby rendering it useless for data stream evaluation, the number one priority should be the development, analysis and implementation of a load shedding technique, similar to the works presented by Tatbul et al. [26] [27]. Other possible, and any less interesting, feature extensions are presented in Table 8.1.

Feature	Feature Description
Load Shedding	An implementation of a load shedding technique for ESC, similar to the works presented by Tatbul et al. [26] [27], could prevent a drop in latency when confronted with very high bursts of data in an unusual long and coherent amount of time.
Multiple Queries	As for now it is only possible to have one query, or scenario, active inside of ESC. For this feature, ESC could be extended to allow the definition and usage of multiple query graphs at the same time.
Processing on the GPU	When performing massively parallel computation, the usage of the GPU is essential in top scoring systems regarding current benchmarks. Techniques and frameworks for using the GPU for processing user functions could be examined, and an adapter could be created which guides as an interface for ESC to do computations on the GPU.
Hot Code Reloading	If the code for the currently active scenario changes, every node needs to be stopped, equipped with the updated code, and restarted. When there are a lot of nodes involved, this approach is not feasible. As Erlang already provides methods for hot code reloading and code reload events, these techniques should be analyzed and implemented into ESC.
Improved Scenario Description	The description of a scenario is currently possible through two steps. First of all, a scenario configuration file needs to be created in an Erlang dependant format. Secondly, the functionality of the processing elements needs to be programmed in Erlang modules using the Erlang programming language. In order to make the system more flexible, the web interface should be extended to allow the user to interactively create and modify scenario graphs. Further on, it should be possible to describe the functionality of a processing element in another programming language, for example Javascript.
Performance through NIF	Erlang provides an interface for the usage of system dependent code, meaning for example modules written in C and compiled for one specific computer architecture. This code is, on the one hand, extremely fast compared to functionality written directly in Erlang, but, on the other hand, mostly platform dependent and hard to write. In a future version of ESC, performance critical parts of the system itself could be programmed with the help of modules written for specific architectures making use of the native interface. Thereby, a huge increase in performance could be achieved.

Table 8.1: ESC Future Work

Bibliography

[1] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The Design of the Borealis Stream Processing Engine. In *Second Biennial Conference on Innovative Data Systems Research (CIDR 2005)*, Asilomar, CA, January 2005.

[2] Daniel J. Abadi, Don Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: a new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, August 2003.

[3] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[4] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB '04, pages 480–491. VLDB Endowment, 2004.

[5] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[6] Irina Botan, Donald Kossmann, Peter M. Fischer, Tim Kraska, Dana Florescu, and Rokas Tamosevicius. Extending xquery with window functions. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 75–86. VLDB Endowment, 2007.

[7] Uwe Brinkschulte, Mathias Pacher, and Alexander Von Renteln. Towards an artificial hormone system for self-organizing real-time task allocation. In *Proceedings of the 5th IFIP WG 10.2 international conference on Software technologies for embedded and ubiquitous systems*, SEUS'07, pages 339–347, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[9] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[10] CISCO. Cisco completes acquisition of truviso. http://shar.es/GeNmR, 2012.

[11] M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *Trans. Sys. Man Cyber. Part B*, 26(1):29–41, February 1996.

[12] Ericsson AB. Erlang. http://erlang.org/doc/man/erl.html, 2012.

[13] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[14] Bugra Gedik, Henrique Andrade, Kun-Lung Wu, Philip S. Yu, and Myungcheol Doo. Spade: the system s declarative stream processing engine. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 1123–1134, New York, NY, USA, 2008. ACM.

[15] Sven Graupner, Artur Andrzejak, Vadim E. Kotov, and Holger Trinks. Adaptive service placement algorithms for autonomous service networks. In Sven Brueckner, Giovanna Di Marzo Serugendo, Anthony Karageorgos, and Radhika Nagpal, editors, *Engineering Self-Organising Systems*, volume 3464 of *Lecture Notes in Computer Science*, pages 280–297. Springer, 2004.

[16] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *25th ACM SIGMOD International Conference on Management of Data (SIGMOD 2006)*, June 2006.

[17] LOFAR. LOFAR | LOFAR. http://www.lofar.org/, 2012.

[18] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[19] MIT. Medusa. http://nms.csail.mit.edu/projects/medusa/, 2003.

[20] MIT. MIT intelligent transportation systems. http://mit.edu/its/mitsimlab.html, 2012.

[21] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In Wei Fan, Wynne Hsu, Geoffrey I. Webb, Bing Liu, Chengqi Zhang, Dimitrios Gunopulos, and Xindong Wu, editors, *ICDM Workshops*, pages 170–177. IEEE Computer Society, 2010.

[22] Franz Rammig, Tales Heimfarth, and Peter Janacik. Biologically inspired methods for organizing distributed services on sensor networks. In Kirstie Bellman, Michael G. Hinchey, Christian Müller-Schloer, Hartmut Schmeck, and Rolf Würtz, editors, *Organic Computing - Controlled Self-organization*, number 08141 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2008. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[23] Ichiro Satoh. Bio-inspired deployment of distributed applications. In *Proceedings of International Workshop on Multi-Agents (PRIMA2004), Lecture Notes in Computer Science (LNCS*. Springer, 2004.

[24] Benjamin Satzger, Waldemar Hummer, Philipp Leitner, and Schahram Dustdar. Esc: Towards an elastic stream computing platform for the cloud. In *4th IEEE International Conference on Cloud Computing (CLOUD'11)*, pages 348–355, 2011.

[25] Mehul A. Shah, Michael J. Franklin, Samuel Madden, and Joseph M. Hellerstein. Java support for data-intensive systems: experiences building the telegraph dataflow system. *SIGMOD Rec.*, 30(4):103–114, December 2001.

[26] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[27] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 159–170. VLDB Endowment, 2007.

[28] The Apache Software Foundation. Apache ZooKeeper. http://zookeeper.apache.org/, 2010.

[29] Hans ulrich Heiss and Michael Schmitz. Decentralized dynamic load balancing: The particles approach. In *Proc. 8th Int. Symp. on Computer and Information Sciences*, pages 115–128, 1993.

[30] QI Yang and Haris N. Koutsopoulos. A Microscopic Traffic Simulator for evaluation of dynamic traffic management systems. *Transportation Research Part C-emerging Technologies*, 4:113–129, 1996.

[31] Stanley B. Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Çetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *IEEE Data Eng. Bull.*, 26(1):3–10, 2003.

[32] Erik Zeitler and Tore Risch. Processing high-volume stream queries on a supercomputer. In *Proceedings of the 22nd International Conference on Data Engineering Workshops*, ICDEW '06, pages 147–, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *PVLDB*, 4(11):1181–1188, 2011.

APPENDIX A

Simulation Results Summary

A.1 Load Distribution

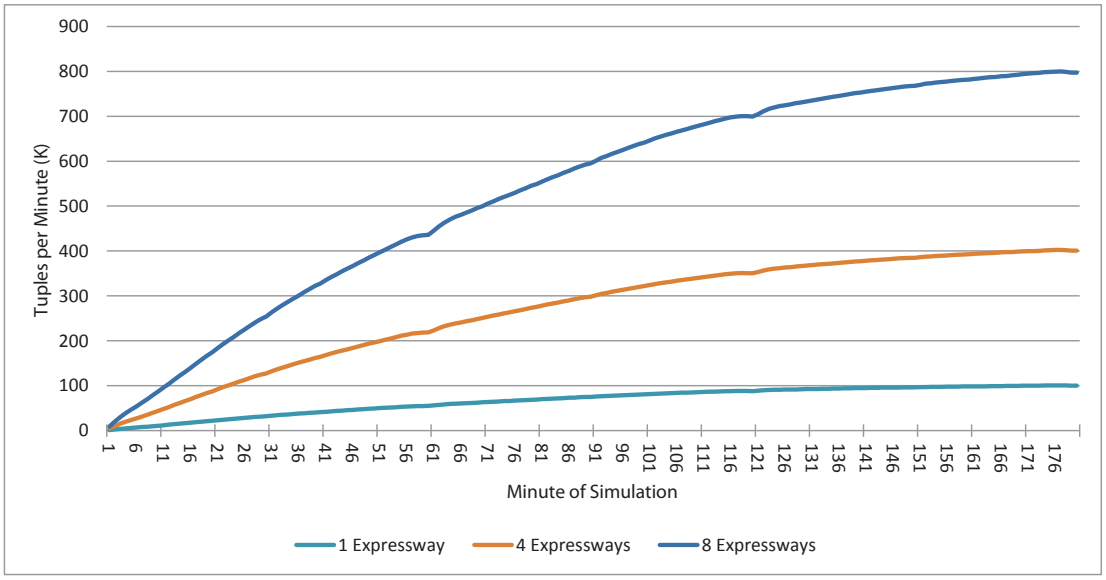


Figure A.1: Load Distribution

A.2 4 Slaves - 4 Expressways

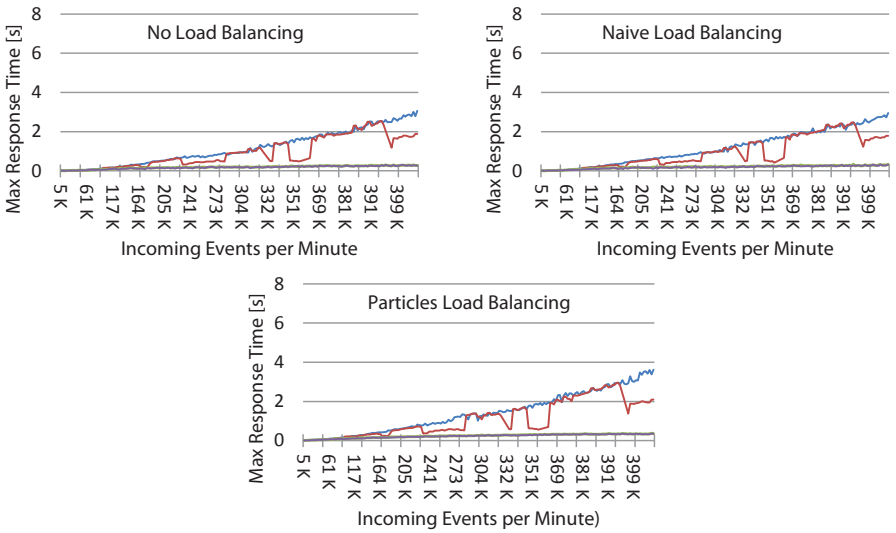


Figure A.2: Similar ID Placement

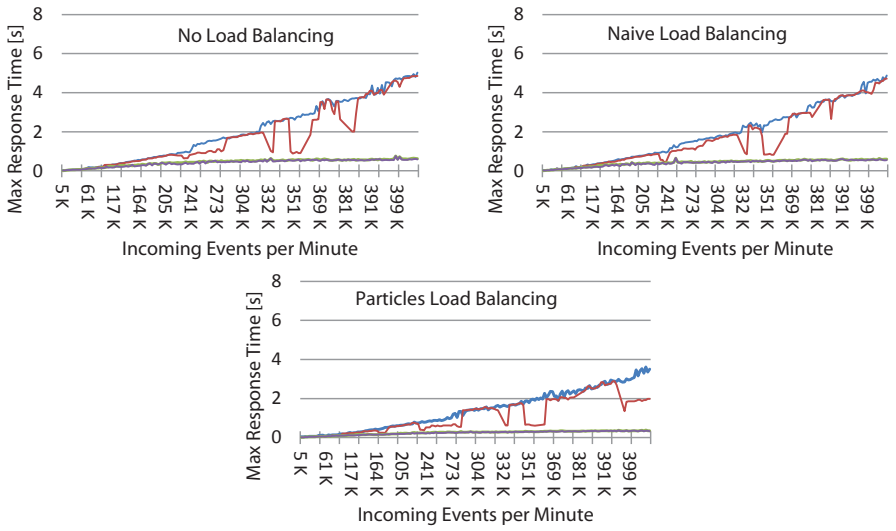


Figure A.3: Random Placement

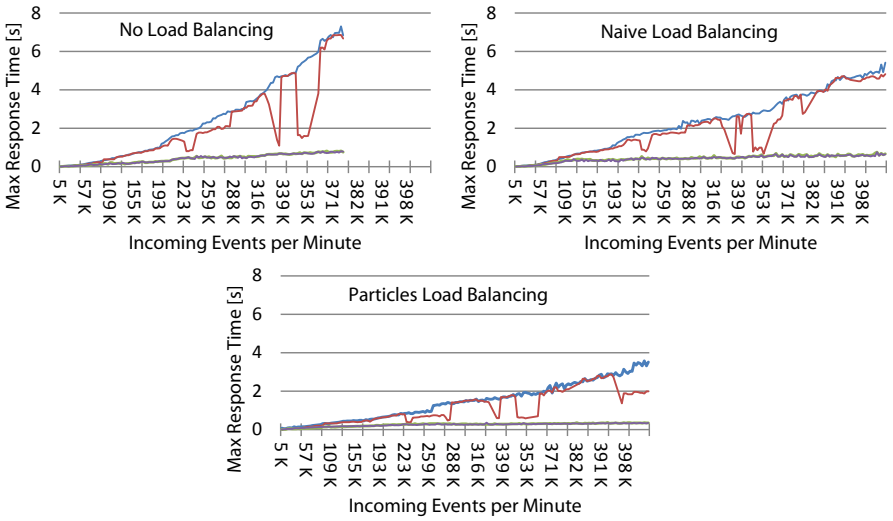


Figure A.4: Least Loaded Placement

A.3 4 Slaves - 8 Expressways

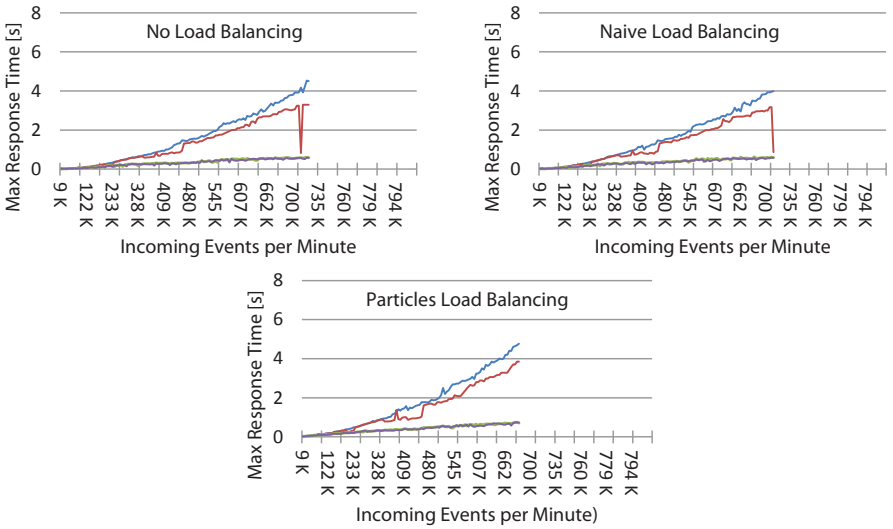


Figure A.5: Similar ID Placement

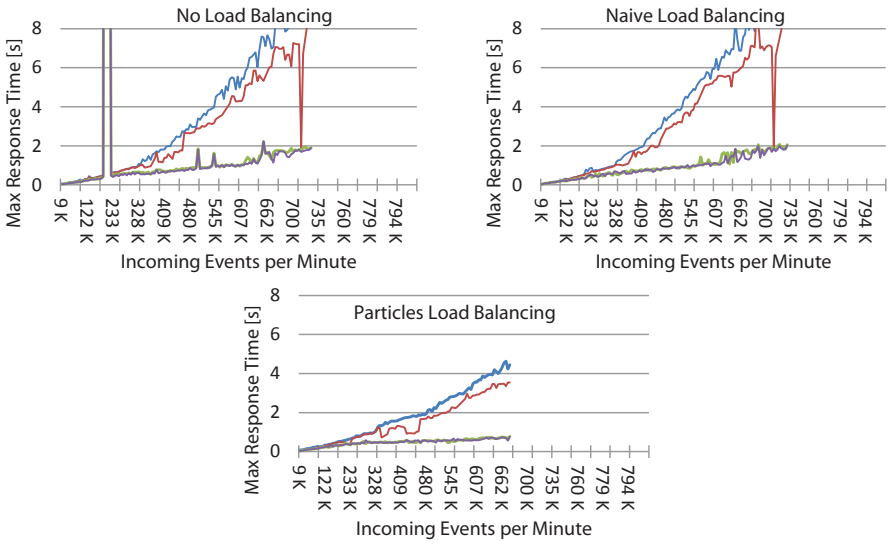


Figure A.6: Random Placement

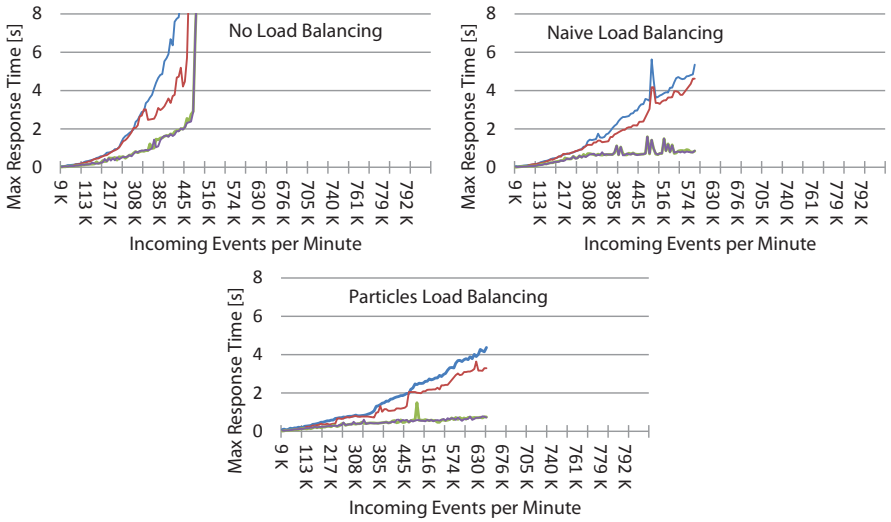


Figure A.7: Least Loaded Placement

A.4 8 Slaves - 4 Expressways

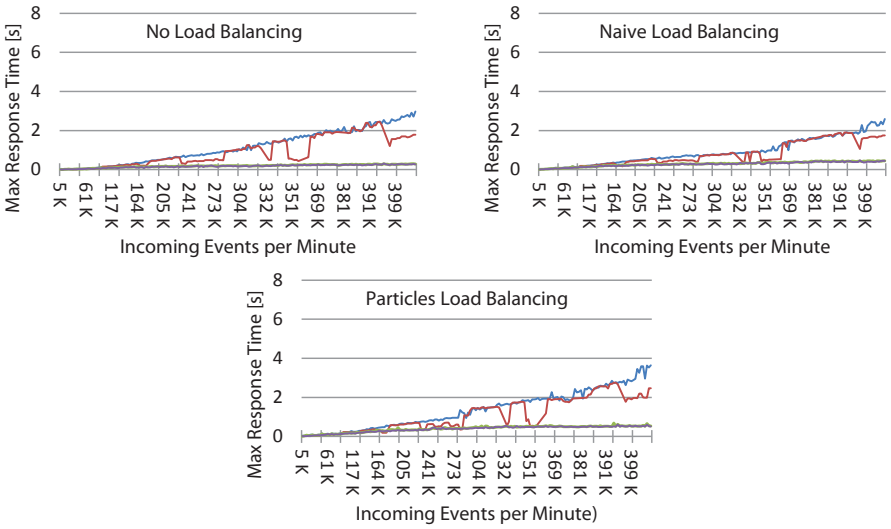


Figure A.8: Similar ID Placement

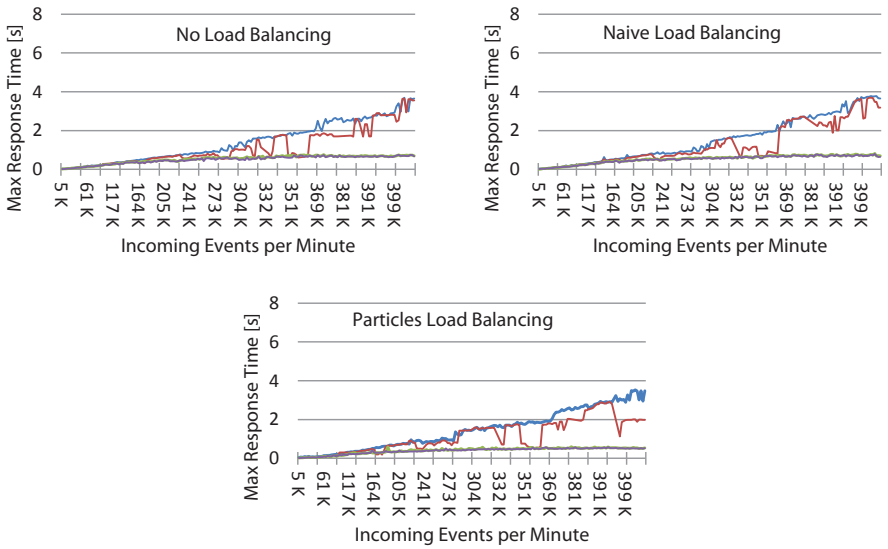


Figure A.9: Random Placement

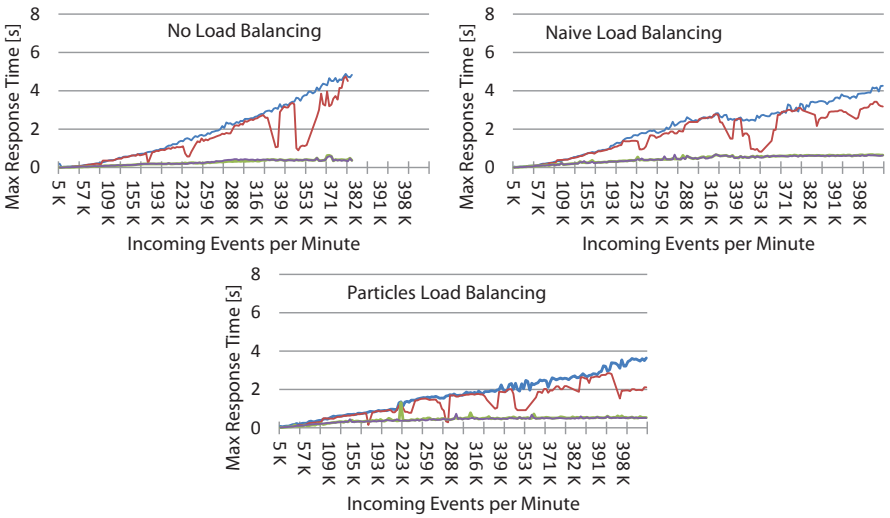


Figure A.10: Least Loaded Placement

A.5 8 Slaves - 8 Expressways

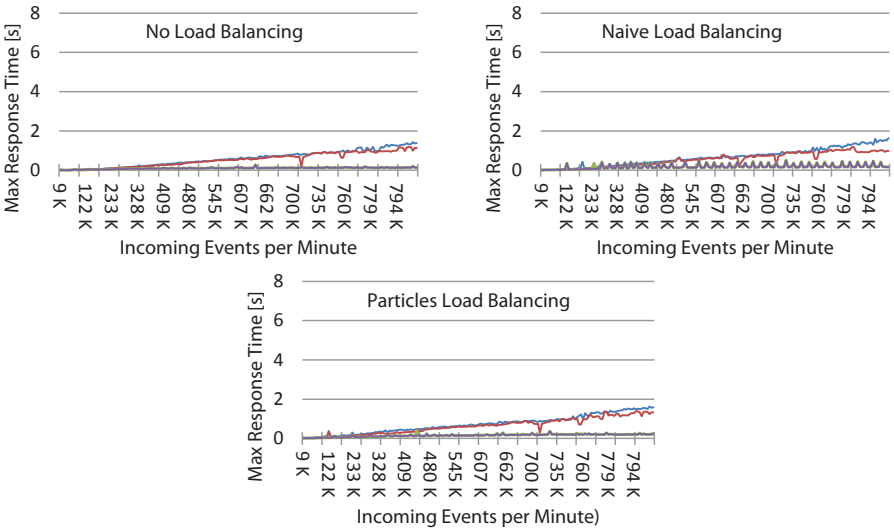


Figure A.11: Similar ID Placement

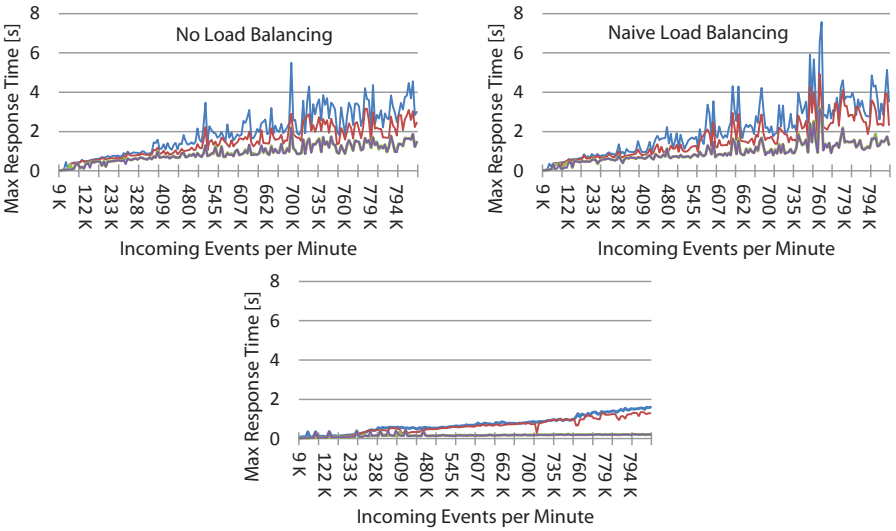


Figure A.12: Random Placement

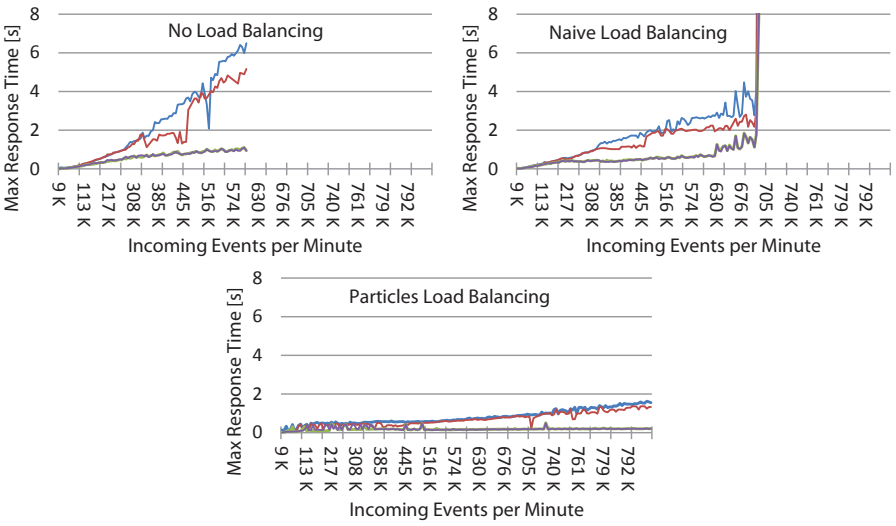


Figure A.13: Least Loaded Placement

APPENDIX B

ESC User Manual

B.1 Introduction

ESC (pronounced “Escape”) is a distributed stream processing platform written in Erlang. It offers a simple programming model in which programs are specified by directed acyclic graphs (DAGs). The following chapters describe the system requirements of ESC, where to get a copy of ESC, the installation procedure and some scenarios, that are shipped with the system.

B.2 Requirements

As ESC runs within an Erlang virtual machine so the requirements for running ESC are identical with the requirements for running the current Erlang release. The current Erlang release can be obtained from `http://www.erlang.org/download.html`. Currently the version R15B01 of Erlang is supported for running ESC.

It should be noted that the system requirements for running ESC are very low. But when it comes to the execution of a specific scenario the requirements to the underlying hardware rise according to the requirements of the scenario under execution. For executing the Linear Road scenario with 0.5 expressways there should be at least one computer with 4 GB memory, a minimum of 4 GB free disc space and a dual core CPU with 2 GHz. Due to the distributed characteristics of ESC there also can be several computers connected to each, which in combination meet the described minimum requirements.

In order to use the ESC monitoring interface, a browser with support for the current web socket protocol (draft-ietf-hybi-thewebsocketprotocol-17), which is described under the URL `http://tools.ietf.org/html/rfc6455`, is necessary. These include at the moment every current WebKit based browser, Opera since version 10.70 and Firefox since version 4.0 beta 7. For development the Google Chrome browser in version 19.0.1084 has been used. In order to view the web interface, a web server software like the Apache HTTP Server or Microsoft IIS is required.

Summary

- 256 MB RAM and 50 MB free hard drive space
- recent Erlang VM (`http://www.erlang.org/download.html`)
- GNU GCC compiler (`http://gcc.gnu.org/`)
- Cygwin (Windows) & GNU Bash (`http://www.gnu.org/software/bash/`)
- Webserver (e.g. Apache HTTP Server or Microsoft IIS Express)
- Browser with support for current web socket protocol (e.g. Google Chrome)

B.3 Installation

First of all the most recent Erlang version needs to be installed, which can be obtained from `http://www.erlang.org/`. For a UNIX based operating system the current version from the package management system should be sufficient. If not already present, a web server software should be installed to view the ESC web interface. It is possible to obtain for example a copy of the Apache HTTP server from `http://httpd.apache.org/` or a copy of the Microsoft IIS Express from `http://www.microsoft.com`. A recent browser supporting the current web socket protocol needs to be installed. The current version of the Google Chrome browser is recommended, which can be obtained from `https://www.google.com/chrome`. If ESC should run on a Microsoft Windows operating system, Cygwin needs to be installed (`http://www.cygwin.com/`). Make also sure that the GNU GCC compiler is installed and available (`http://gcc.gnu.org/`) on your system.

ESC can be pulled from `gitvienna.vitalab.tuwien.ac.at:esc.git` to a folder of your choice. Afterwards the web server needs to be configured to point into the folder `%ESC_HOME%/priv/www/` for viewing the ESC web interface. If a Microsoft Windows operating system is used, Cygwin must be started. For every other operating system a console (GNU Bash is recommended) must be opened. It has to be ensured that the opened console points to the ESC installation directory. By executing the command `bash start_master.sh` the master node is started to which all slave nodes can connect from this point on. The ESC interface can be accessed by opening the URL `http://localhost/index.xhtml` in the installed web browser (depending on your webserver configuration). Within another console window a slave node can be started by executing the command `bash start_slave1.sh`. Attention: It is not recommended to start the master and the slave application on the same computer when executing scenarios with high computational requirements! In order to connect other machines from the network it is necessary to adjust the IP addresses inside the scripts `start_master.sh` and `start_slave1.sh` accordingly.

Summary

- Erlang installation (`http://www.erlang.org/`)
- installation webserver and browser (with support for the web socket protocol)
- pull copy of ESC (`git@vienna.vitalab.tuwien.ac.at:esc.git`)
- point the web server to the folder %ESC_HOME%/priv/www/
- open Cygwin console (Windows) or GNU Bash and go to the ESC installation directory
- execute command `bash start_master.sh`
- open another console window and execute command `bash start_slave1.sh`
- open the URL `http://localhost/index.xhtml` inside the browser (URL depends on your webserver configuration)
- for connecting other nodes from the network it is necessary to adjust the IP addresses in the scripts `start_master.sh` and `start_slave1.sh` accordingly

B.4 Configuration

ESC can be configured by the scripts `start_master.sh` or `start_slave1.sh` respectively. The possible command line arguments are presented in the following two tables.

start_master.sh

Argument	Description
-d	When this parameter is given then the application is compiled in debug mode before running it. Every debug output is saved into the file `log/esc.log`. As debug output puts a huge amount of load on the system, this setting should be used with care.
-c	When this parameter is provided the application is compiled before it is started.
-m NODE	The name under which the master node should be reachable. The default parameter is `master127.0.0.1`. It should just be necessary to adjust the IP part of the name, either via the command line argument or directly inside the script.

Table B.1: Parameter for the Master Application

start_slave.sh

Argument	Description
-m NODE	Name of the master node to which the slave will connect. The default value is `master127.0.0.1`.
-s NODE	Name of the slave node under which the slave node will be reachable. This name should be unique under all slave nodes. The default name is `slave1127.0.0.1`.
-d	Starts the slave in a "detached" state, which means it cannot be controlled via the command line anymore.

Table B.2: Parameter for the Slave Application

B.5 The ESC-Web-Interface

ESC can be controlled either directly via the command line, or via the web interface. As mentioned earlier if you want to use the web interface it is necessary to have a recent browser installed, which supports the web socket protocol, as well as a web server that points to the location of the folder containing the `index.xhtml` file inside the ESC installation directory. If everything has been setup correctly, you should be able to see the web interface by accessing `http://localhost/index.xhtml` (depending on the configuration of your web server).

Web Sockets

The usual approach when writing client applications was to poll the server for the availability of new data to update the client view. This approach has the advantage that the server does not have to keep many connections open and no state needs to be preserved. The downside is that due to the polling a huge computational overhead is created on the server side. Especially when dealing with stream processing applications and their monitoring a polling approach is not desirable as it could slow the system down immensely.

With the use of web sockets a permanent connection is established between the browser and the server, in which both sides are able to initiate data transfers (push approach). By using this technique, the server application is able to push information, e.g. load information that he received from his slave nodes, to the client browser. The advantage is a huge speed increase on the client as well as an on the server side. As of now there can be only one active connection to the server application, due to the yet missing connection management.

More information on web sockets can be obtained by reading the respective Wikipedia article (`http://en.wikipedia.org/wiki/WebSocket`) or the RFC (`http://tools.ietf.org/html/rfc6455`).

Web Interface

When accessing the ESC web interface for the first time, the home view shown in Figure B.1 is displayed. At the top of the page you can see the navigation bar together will a status field. Within the status field messages regarding the communication with the server are displayed. Whenever there occurs an error or no connection to the server could be established, the messages appear in red, all other messages appear in green. The home view gives information about the available sections, which will also be described in the following chapters.

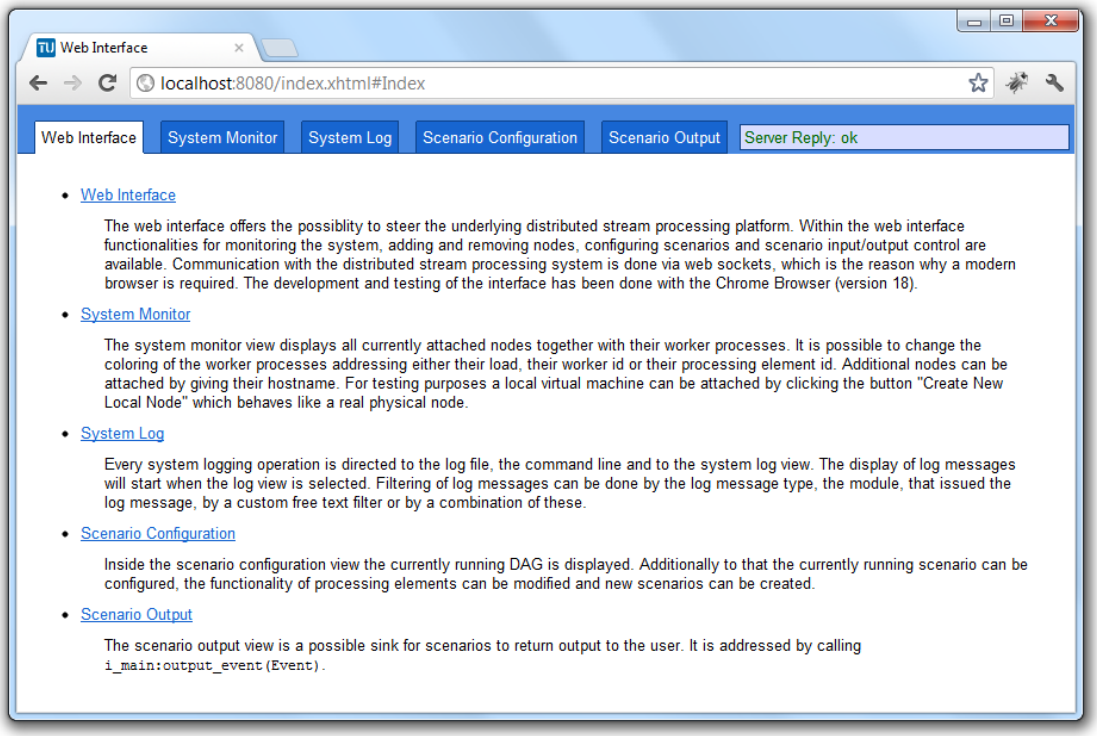


Figure B.1: ESC Web Interface - Home

System Monitor

The System Monitor view, which is shown in Figure B.2, shows all nodes currently connected to the master and their worker. Within the System Monitor view it is possible to connect new nodes (which should not be necessary, as slave nodes connect themselves automatically), create new local nodes and remove (detach) nodes. The creation of new local nodes involves the initialisation of a detached slave instance on the machine of the master node and should only be used for debugging or testing purposed.

Another feature is the monitoring of workers currently running on a specific node. This feature can be activated or deactivated by clicking “Activate Worker Monitoring” or “Deactivate

Worker Monitoring" respectively. It is not recommended to use worker monitoring when dealing with more than 1000 workers per node as it could slow down the system immensely and therefore worker monitoring is per default deactivated.

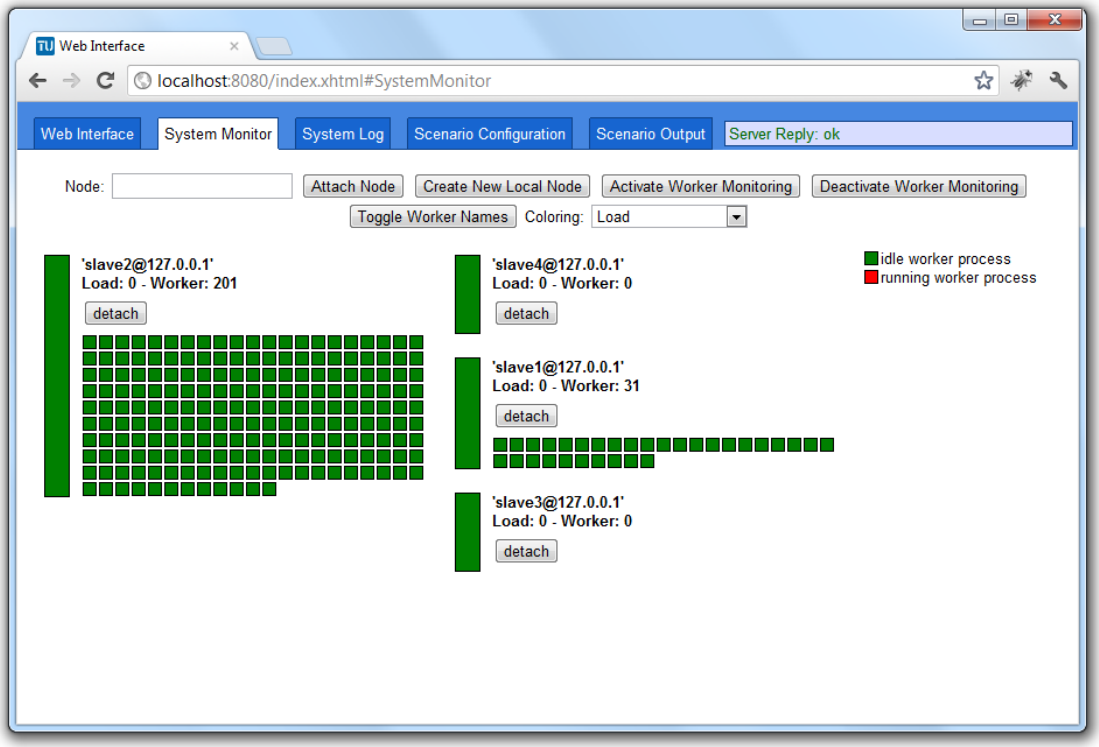


Figure B.2: ESC Web Interface - System Monitor

With worker monitoring activated, several color patterns can be applied, defining the colouring of the worker. The currently available color patterns include "Load", "Same Worker Key" and "Same Pe-Id". An overview of the available colourings and how they work is given in Table B.3 and an example of the "Same Pe-Id"-colouring is given in Figure B.3.

Load	A worker process is coloured by its load (messages in his queue). When its queue length is bigger than zero, its colour is red. If no messages are in its message queue than its colour is green.
Same Worker Key	The colouring of the worker processes is based on the worker key. Whenever a new worker key occurs a new random colour is assigned to that worker key.
Same Pe-Id	The colouring of the worker processes is based on the Pe-Id. Whenever a new Pe-Id occurs a new random colour is assigned to that Pe-Id.

Table B.3: Worker Coloring Patterns

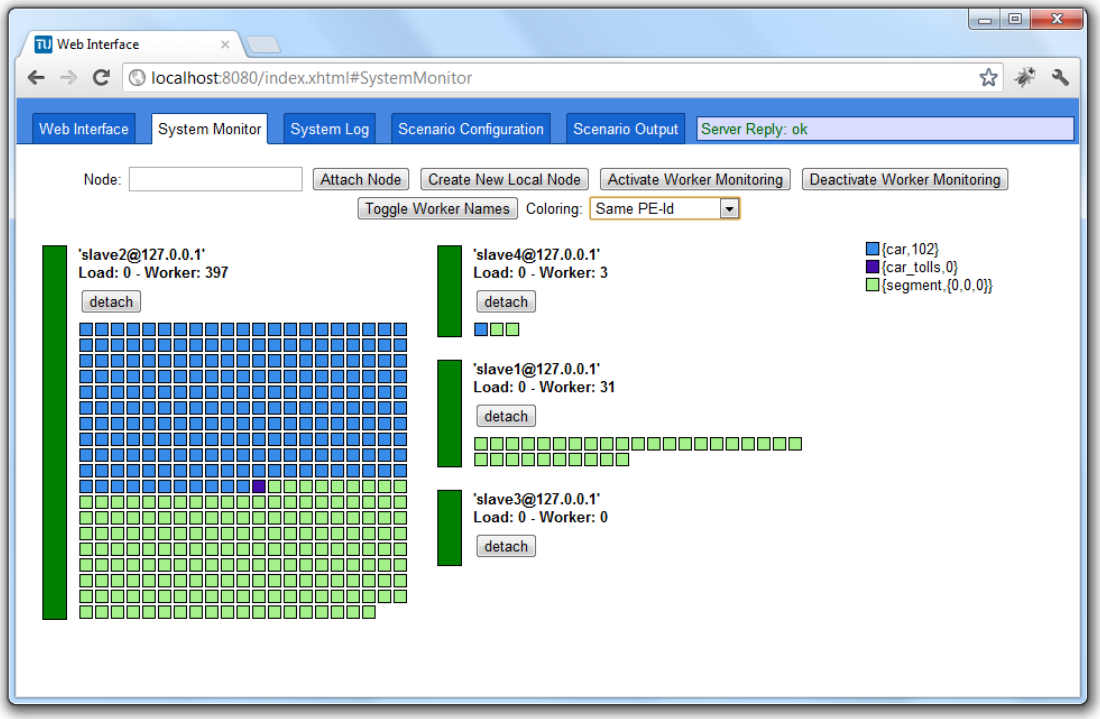


Figure B.3: ESC Web Interface - System Monitor with Coloring

System Log

Within the System Log view (depicted in Figure B.4) every log output of the application can be traced and filtered. To prevent unnecessary load and bandwidth consumption, the log entries are only updated whenever the System Log view is selected. It is possible to filter the log output by the severity (debug, info, warn, error), by the module name or by a custom string. A combination of all three filters is also possible. Whenever new log entries occur, the browser automatically scrolls down to keep the new entries visible, but this feature is only enabled when the scrollbar is on the bottom position.

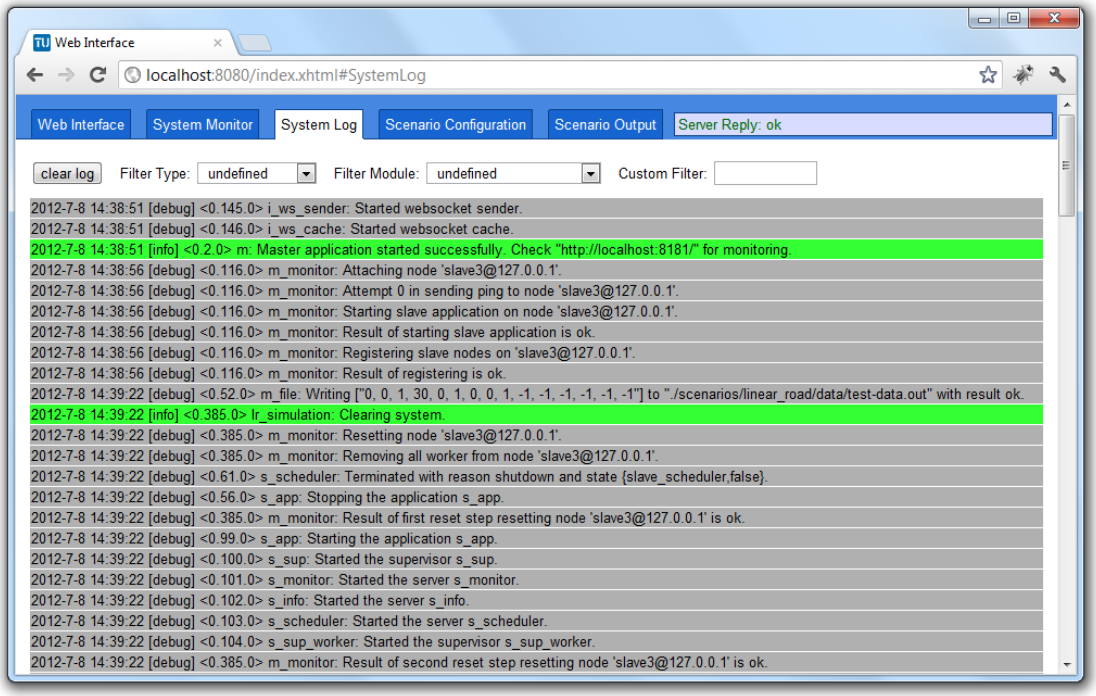


Figure B.4: ESC Web Interface - System Log

Scenario Configuration

The Scenario Configuration view displays the scenario graph of the currently loaded scenario. Additionally to that, the active scenario can be changed and specific events can be fired. The list of scenarios is derived from the scenarios available inside of the scenario folder within the ESC installation directory.

In future versions of ESC it should be possible to create new scenarios or to modify existing scenarios within the web interface. By clicking on a node the currently active functionality of the specific node shall appear and it should be possible to adjust the code either in Erlang or in another programming language of your choice.

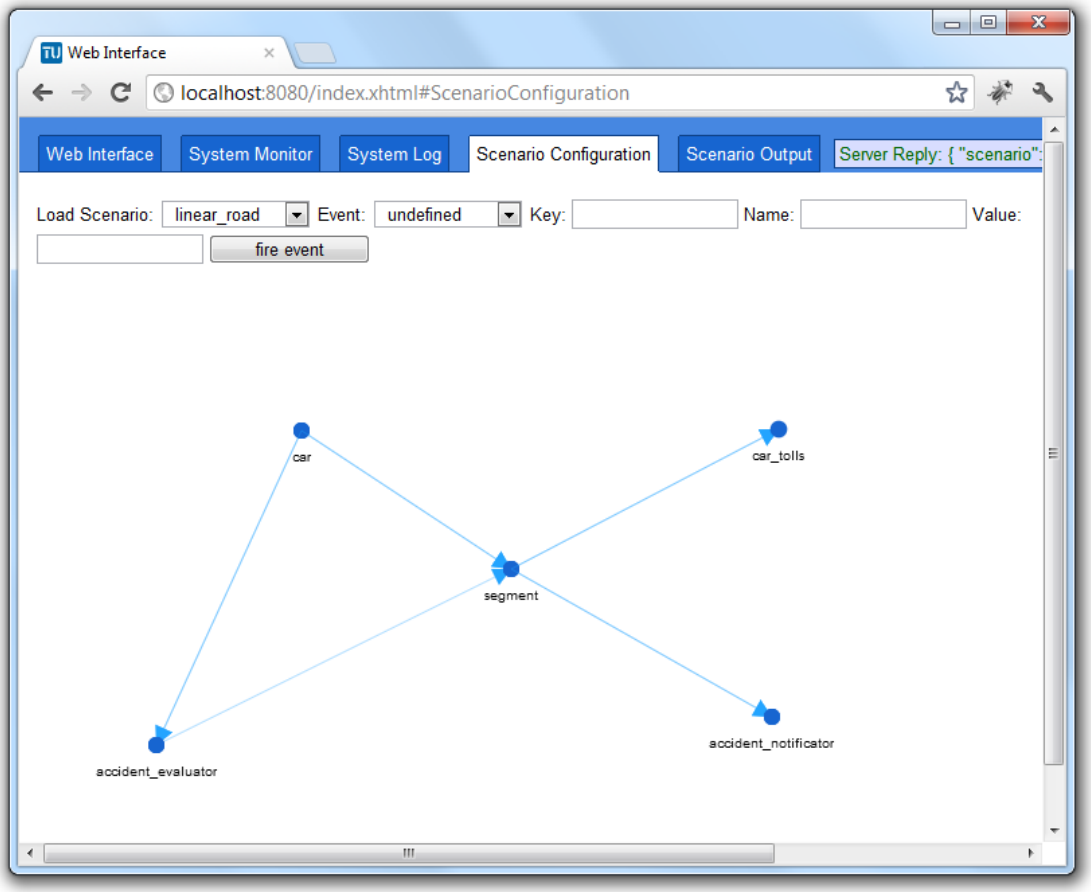


Figure B.5: ESC Web Interface - Scenario Configuration

Scenario Output

The Scenario Output view can be used by the running scenario to publish its results to the web interface. This can be achieved by calling `i_main:output_event(Event)`. Afterwards the event should be visible within the web interface.

B.6 Creating and Running Scenarios

In order to create your own scenario for ESC the following steps are necessary:

- Create an new folder inside the scenarios folder with your scenario name as folder name
- Place a new file inside the folder with a name of the form `scenario_name.config`
- Adjust the contents of the configuration file according to your needs (see Chapter B.5)

- Place your source files inside the scenario folder into a sub folder of your choice (be careful not to take a file name or module name that has been taken by any other module or file within the ESC application, including all other scenarios)
- Start ESC by executing the command `bash start_master.sh -c`
- Load your scenario by executing `m:load_scenario(scenario_name).`
- The scenario is now running and can receive events

B.7 Create scenario configuration file

The scenario configurations file contains three sections: a graph section, a processing element description and a plug-in section. The following example from the Linear Road scenario shall give an overview of the possible settings and their meaning.

```
{graph, [ % scenario graph as a list of adjacent PE-Ids
    {xway0, [car]}, {car, [accident_evaluator]},
    {accident_evaluator, []}
]}.

{pe_ids, [ % PE-Ids with module, arguments and options
    {xway0, [
        {module, lr_xway}, {arguments, []},
        {events, any}, {node, any} {hibernate, off},
        {movable, false}
    ]}
]}.

{properties, [ % global scenario properties
    {load_balancing_strategy, lb_none},
    {placement_strategy, pl_least_loaded}
]}.
```

Listing 12: Scenario Configuration

B.8 Example Scenarios

Some example scenarios come together with ESC and can be found inside the folder scenarios. These scenario include the demo scenario, which basically used to do unit testing, and the Linear Road scenario which implements the Linear Road benchmark. The Linear Road benchmark has been developed to test the performance of stream data management systems by simulating the traffic on a specific amount of expressways. More information concerning the Linear Road benchmark can be found under `http://pages.cs.brandeis.edu/~linearroad/` and in Chapter B.8.

Demo Scenario

The Demo scenario is used mainly for debug purposes and unit testing. In order to generate load inside the processing elements each element generates a certain amount of prime numbers. For the consumption of memory random strings are created and saved in the state of the worker. The demo scenario contains the following processing elements:

input	Used for receiving incoming events.
non_movable	A processing element that is not allowed to move between nodes.
heavy_non_movable	Also a processing element that is not allowed to move, like non_movable, but additionally this processing elements contains large strings in its memory.
small	A processing element with a low memory and load consumption.
heavy	A processing element with high load and memory consumption.
output	The processing element used for receiving output events.

Table B.4: Processing Elements for Demo Scenario

Linear Road

As described before, the Linear Road benchmark simulates the traffic on a specific amount of expressways. The exact specifications for the Linear Road benchmark can be observed in the paper located under `http://www.cs.brandeis.edu/~linearroad/linear-road.pdf`. In order to start the Linear Road benchmark input data is needed. This input data can be generated as described under `http://pages.cs.brandeis.edu/~linearroad/mitsiminstall.html`.

After the generation of input data for a specific amount of expressways, the simulation can be started by executing the command lr_simulation:start("history_input", "stream_input"). After a period of three hours the simulation is finished and all results are written into the output folder of the Linear Road scenario folder.