Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Efficient Splitter for Data Parallel Complex Event Procesing

Marco Amann

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Dr. Kurt Rothermel
Supervisor:	M. Sc. Ahmad Slo
Commenced:	March 19, 2018
Completed:	September 19, 2018

Abstract

Complex Event Processing systems are a promising approach to detect patterns on ever growing amounts of event streams. Since a single server might not be able to run an operator at a sufficiently high rate, Data Parallel Complex Event Processing aims to distribute the load of one operator onto multiple nodes. In this work we analyze the splitter of an existing CEP framework, detail on its drawbacks and propose optimizations to cope with them. This yields the newly developed SPACE framework, which is evaluated and compared with an industry-proven CEP framework, Apache Flink. We show that the new splitter has greatly improved performance and is able to support more instances at a higher rate. In comparison with Apache Flink, the SPACE framework is able to process events at higher rates in our benchmarks but is less stable if overloaded.

Kurzfassung

Complex Event Processing Systeme stellen eine vielversprechende Möglichkeit dar, Muster in immer größeren Mengen von Event-Strömen zu erkennen. Da ein einzelner Server nicht in der Lage sein kann, einen Operator mit einer ausreichenden Geschwindigkeit zu betreiben, versucht Data Parallel Complex Event Processing die Last eines Operators auf mehrere Knoten zu verteilen. In dieser Arbeit wird ein Splitter eines vorhandenen CEP systems analysiert, seine Nachteile hervorgearbeitet und Optimierungen vorgeschlagen. Daraus entsteht das neue SPACE Framework, welches evaluiert wird und mit Apache Flink, einem industrieerprobten CEP Framework, verglichen wird. Wir zeigen, dass der neue Splitter ein bedeutend größeres Durchsatzvermögen besitzt, sowie mehr Instanzen mit einer höheren Rate versorgen kann. Im Vergleich zu Apache Flink kann das SPACE Framework in unseren Experimenten höhere Durchsatzraten erzielen, ist aber nicht so stabil, wenn es überladen wird.

Contents

List of Figures

List of Listings

1 Introduction

Since ever growing numbers of interconnected sensors, devices and services are deployed, the amount of data produced increases [Cor14]. This requires new methods to process the data collected or generated at high speeds to create valuable high-level information that is useful for consumers. A typical example is the detection of real-world scenarios based on sensor readouts [MSPC12]. Depending on the use-case, these systems may require near-realtime processing of their data to enable consumers to act upon it. Furthermore, it might be required that data collected from multiple sources is combined and analyzed in order to detect the desired results.

An example for a system with these requirements is one that is capable of analyzing the large amounts of data produced by interconnected sensors in IoT scenarios, since these systems reveal their real value, if data from multiple sources is correlated and analyzed [BDR+16]. Furthermore, IoT systems may require processing and acting in a timely manner to be truly useful [WLLB06]. Other areas of application might be the processing of log messages or high frequency trading.

Some of these systems may require low latency processing of an incoming stream of data, rendering established patterns like batch-processing, the accumulation of data and its subsequent processing in large batches, unsuitable. Therefore it gets more and more important to be able to directly process queries on a continuous stream of data without long-term storage of produced information.

A promising approach to cope with such amounts of data under these constraints is Complex Event Processing (CEP), where incoming data is processed as a stream of events and these events are matched against queries deployed to the system. CEP is capable of analyzing queries on multiple input streams and produce higher order, complex events as an output stream. These events can again be fed to parts of the system to generate new complex events of even higher order [SS13]. This enables systems to decide based on real-world events, like a fire in a factory building, by correlating primitive events, like temperature measurements and data provided by smoke detectors, each containing only a small amount of partial information.

A typical CEP system is comprised of sources that collect or produce data, operators that analyze the data streams and sinks, that consume the data. The topology of a CEP system defines how the operators, sources and sinks are connected. If the rate of events increases, an operator running on a single compute node might not be sufficient to process all incoming events, hence the need to distribute the load onto multiple machines arises. Despite being able to have multiple operators, which execute queries, a query can still require one operator to process all incoming messages and therefore can be the bottleneck of the system. This limitation can be approached by Data Parallel CEP, which enables multiple copies (instances) of the same operator logic to be run in parallel. These operator instances do the same work but on different data. The distribution of events to these instances and the ordering of incoming events from source-streams is handled by a single component, the splitter.

The splitter in a CEP operator has to cope with the full event rate and therefore needs to be able to efficiently order and distribute incoming events, to be able to fully exploit the benefits of parallelization. Therefore, a performant and stable splitter is crucial for the overall system performance.

In this thesis, we analyze the splitter component of PACE, a framework for Data Parallel CEP, and propose possible improvements. These improvements are then implemented and form the SPACE framework. We evaluate the effects of our improvements and compare the new framework to an open source framework capable of complex event processing, Apache Flink.

The remaining part of the thesis is structured as follows: First, we introduce common concepts of Complex Event Processing found in literature in Chapter 2 and illustrate the importance of the splitter component in a Distributed Complex Event Processing system. After the theoretical foundations are outlined, we describe the PACE framework in Chapter 3 and detail on its implementation as well as some of its shortcomings. To cope with these problems, we present possible improvements in Chapter4. In Chapter 5, we evaluate the SPACE framework and compare it to Apache Flink. We then summarize the work done and outline possible future work in Chapter 7.

2 Background

In this chapter we illustrate common approaches and systems to analyze streams of events and focus on the theoretical foundations underlying modern CEP systems.

2.1 Batch and stream processing

To process a stream of data, one can store this data and process it in batches. This so called batch processing is a widely used practice, for example supported by Hadoop [Tay10]. When dealing with a batch of data, the amount of data is finite at the time of processing. This introduces latency but simplifies processing logic. To illustrate this, it is trivial to compute the average of a given set of numbers, compared to computing the average of an endless stream of numbers. There are use cases where storing events for hours or even days might not be viable, either because it is simply too much data to store or the data has to be processed within short time constrains to enable decisions with low latency. In such cases, one can deploy stream processing to handle data as it arrives. There exist numerous frameworks that support such processing like Apache Flink [1], Twitter Heron [2] or Esper [3].

2.2 Complex Event Processing

Complex Event Processing is a special approach to process data streams. While stream processing focuses on applying transformations on input streams like averages or minima, Complex Event Processing focuses on detecting patterns on a stream of primitive events to produce a stream of higher level complex events [CM12].

2.2.1 Key concepts of Complex Event Processing

In this chapter we introduce some concepts underlying CEP systems, commonly found in literature. Later, these concepts are explained in depth in their respective sections.

[1] https://flink.apache.org/
[2] https://apache.github.io/incubator-heron/
[3] https://www.espertech.com/esper/

Events An event e represents some change of state in a system, it does not incorporate any duration but rather happens at a point in time [Hed17]. An event consists of its payload, for example a sensor readout or changes in stock data. An event further consists of some metadatai, for example timestamps or serial numbers. An event is of a specific type. The type-information has to be saved within the event.

Event Stream An event stream l connects two endpoints. It's a collection of events $e_{0 \ldots n}$, where the index specifies a serial number generated in the source of the event, that is unique for the stream. This stream can be unbounded [KMR+13]. There can be multiple types of events in one stream [NRNK10], for example events of users clicking a web-page and events of the webserver answering the requests.

Window A window w is a subset of an event stream l and hence partitions the stream in sections $w \subset l$ [MTR17]. A window has a start and end predicate P_{start} and P_{end}, that open and close the window. Hence the window has a finite amount of events $(e_a, e_{a+1}, \ldots, e_{a+m})$.

Complex events One or more complex events c can be triggered when evaluating a window w. A correlation function $f : w \rightarrow (c_0, \ldots, c_n)$ defines how complex events are produced from windows [MKR15]. Note that a single window can produce multiple complex events.

Order There exists an order $<_e$, between all events of all streams. All events in a stream are ordered according to their increasing serial numbers, such that $\forall (e_i, e_j) \in l : i < j \Rightarrow e_i <_e e_j$. Since these serial numbers are unique and increasing per stream, the events of one stream are automatically ordered by timstamp.

When merging multiple streams L, the events need to be ordered in a way such that the following condition holds, with $t(e)$ representing the timestamp of e:

$$\forall l_a, l_b \in L : \forall (e_i, e_j) \in l_a \times l_b, t(e_i) < t(e_j) : e_i <_e e_j$$

If there are multiple events present with the same timestamp, a tie-breaker can be deployed. This tie-breaker can be based on the IDs of the individual streams or the serialnumbers within the originating streams.

2.2.2 Components of a CEP system

In this section we describe a typical CEP system as depicted in figure 2.1 [Hed17], [MTR17], [CM12].

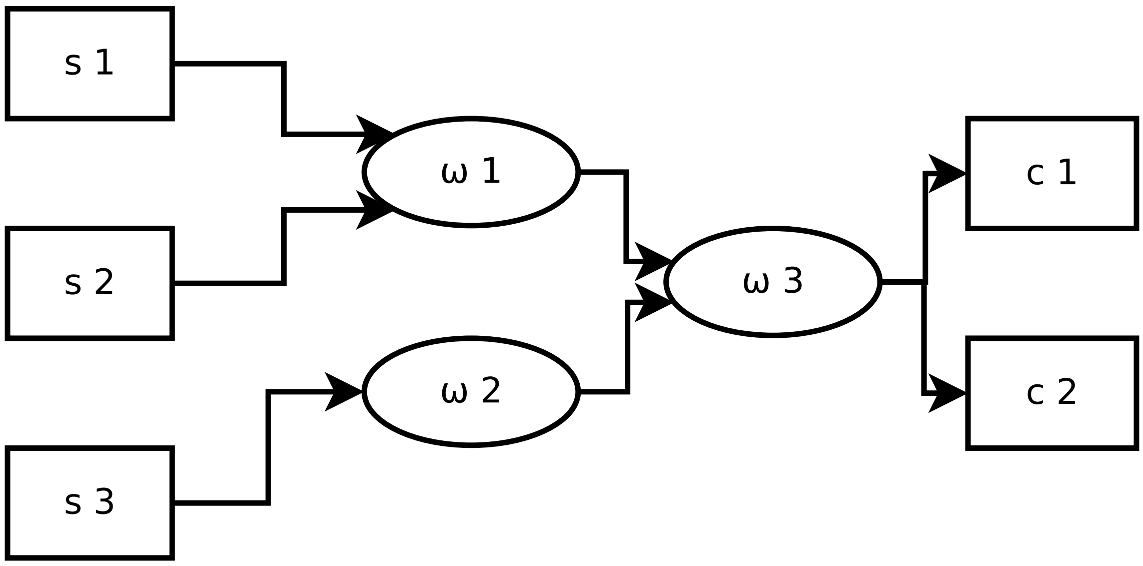


Figure 2.1: Sample operator graph as depicted in [MTR17]

Sources The sources S of a CEP system, as depicted in figure 2.1, gather or produce the events that will be processed by the system. The sources are connected to an operator, whereas one operator can have one or more connected sources. The information originating from a source $s_i \in S$ is an event stream $l_i \in L$. Examples for typical sources are:

- A temperature sensor periodically sending the measured temperature
- A CCTV-Camera sending video-frames as events
- A webserver sending its access-logs as a stream of events

As the examples show, a source can send events at a regular time interval, like CCTV-Camera with 30 frames per second, or at random times, like a webserver that is used by a human. It is also worth noting, that the rates of events can severely differ between sources.

Operator The operator ω, see also figure 2.1, scans the incoming event streams for patterns matching a query. If a match is detected, the operator generates one or more complex events according to its correlation function, which get sent out to its output stream. The set of all operators Ω forms an operator graph $G(\Omega \cup S \cup C, L)$, in which the nodes are interconnected via event streams $L \subset (S \cup \Omega) \times (\Omega \cup C)$ [KMR+13].

In Distributed Complex Event Processing, multiple operators can be deployed onto several machines to spread out the load onto multiple machines [MTR17].

Sinks The sink $c \in C$ is a consumer interested in the produced complex events. This component can forward received complex events to other applications acting upon the received information. Common CEP systems support sinks forwarding data to RDBMS or NoSQL systems [Fou18c].

2.2.3 Shortcomings of centralized CEP systems

When a centralized CEP system has its operator graph deployed on a single compute node, the resources of this machine might not be sufficient to process the whole stream of incoming events, thus imposing a limitation on the overall system throughput. This might be caused by a sufficiently high rate of events or expensive computations on the operator, for example pattern detection on streams of images.

Furthermore when deploying all the operators on one node, the effects of this node failing are much worse than when operators are distributed across multiple nodes.

While being able to place operators onto multiple machines, a single operator might still be overloaded if the structure of the query requires it to process a high enough amount of events. Therefore, it might be of advantage to split the inner workings of the operator itself and enable parallel processing of the event stream. In the next section we detail this approach.

2.3 Parallel CEP

To distribute the load of a single operator onto multiple nodes or CPU cores, parallel CEP is a commonly used practice [MTR17]. In this section we present different forms of parallelization.

As stated by Balkesen et al. [BDWT13], one can parallelize the processing of the event stream by either the states of the finite state machine (FSM) underlying the query, so called state-based parallelism or by splitting the incoming stream of data itself, referred to as Data parallel CEP. The trade-offs between these two approaches are outlined below.

2.3.1 State-based parallelism

With state-based parallelism, the states of the FSM defined from the query are split into multiple processing units [SGN+11]. These processing units can be parallelized under certain circumstances, but this approach has several shortcomings as described by Mayer et al. [MKR15] and Balkesen et al. [BDWT13]:

Communication overhead Partial matches from states early in the detection process have to be transferred to later ones, even if the following state will not match. This introduces additional load on the connections between the nodes running their detection step and requires more computations on the nodes [BDWT13].

Query dependence Since the distribution of the states is based on the FSM, the parallelization degree is depending on the query. This causes some queries to be processed faster than others [BDWT13]. An example for a query that cannot be parallelized at all, is one with a FSM, that has only one state. Furthermore, every FSM has a start state, so states closer to this start state are more likely to be evaluated. As an example, when detecting sequence ‘ABC’, the state detecting C is less likely to be hit than the one detecting A, when the contents of input events are randomly chosen.

Replication of the input stream to every operator. Since every event can trigger a transition of the FSM, every event has to be re-transmitted to every active operator [BDWT13], [MTR17].

Unequal load distribution In addition to the dependence on the query in terms of the evaluation rate, the query might also introduce unequal computational requirements in the states. An example to illustrate this is a system, that first has to perform computationally heavy image recognition and then, in another state, compare two integers.

2.3.2 Data Parallel CEP

In Data Parallel CEP, multiple copies of the operator logic, so called operator instances, process events assigned to them in parallel. These operator instances perform the same task of detecting a pattern but on different data [BDWT13], so the degree of possible parallelization is not based on and therefore not limited by the structure of the query.

Figure 2.3 depicts a typical setup of the operator instances connected to the splitter and merger. These operator instances have the same operator logic but are applying it to different data, that gets sent to them by the splitter.

2.3.3 Splitting

To assign events to operator instances, one has to decide what incoming events have to be processed by which operator instance without producing false positives or false negatives. Two approaches to achieve this are described in the next section in detail.

Key-Based splitting

When determining which events have to be sent to an operator instance, this can be done solely based on information contained within the event, for example an ID used as key. A scenario this may be used in, is a multi tenant system, where events produced by multiple customers have to be processed within a single system and the events of different customers do not correlate at all. In such a scenario, splitting based on a customer-ID attached to an event might be feasible because detecting pattern on the streams can be carried out independently by operator instances. The parallelization degree is limited by the identified semantically independent keys to split upon [BDWT13].

Window-Based splitting

Another approach to split the incoming stream of events at the splitter of an operator is to rely on predicates, that partition the stream of events into windows based on various, user defined parameters. These can be the content of each event, the state of the splitter (e.g. number of processed events so far), the timestamps attached to the events or the current time of the splitter as well as information of the current or other windows, just to name a few. The predicates can be based on a completely user-defined logic, for example the opening and closing of railway barriers when working with the video-feed of a camera at a railroad crossing.

Events can be part of multiple windows. This overlap o of two windows w_1 and w_2 is defined as $o = w_1 \cap w_2$.

Since the amount of events in a window is finite, one can not only detect patterns of events but also their absence. To illustrate this case, think of a query that wants to have a complex event produced, when there is a heartbeat event missing within a 5 minute time window.

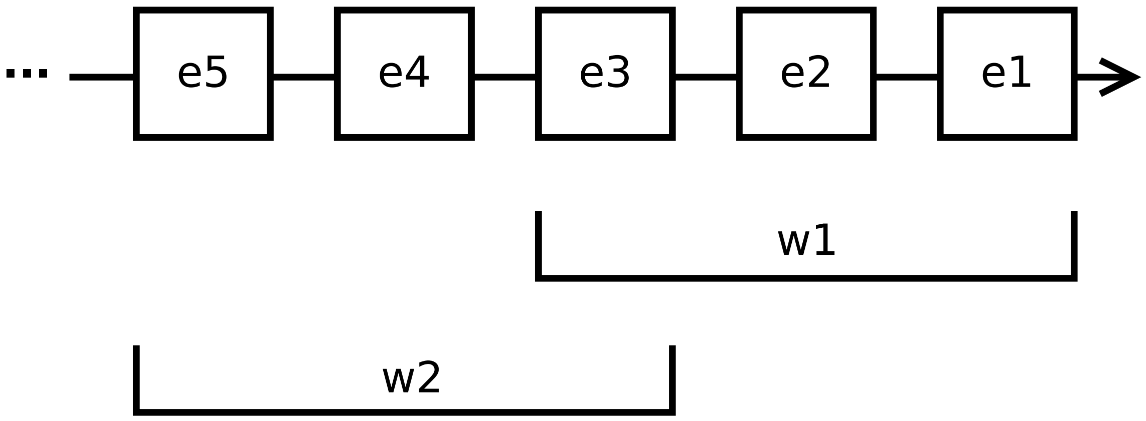


Figure 2.2: Events partitioned by windows

To illustrate the window based splitting, figure 2.2 depicts an event stream containing some events and shows the partitioning into windows. These windows can be independently processed on operator instances (see section 2.3.4). If we assume w_1 get scheduled on operator instance ω_1 and w_2 gets scheduled on operator instance ω_2, events e_1, e_2, e_3 have to be sent to ω_1 and e_3, e_4, e_5 have to be sent to ω_2. Note that e_3 is part of both windows and hence has to be sent to both operator instances. In this case we have an overlap of 1, with a window size of 3, this leads to a window overlap of $^1/_3$.

2.3.4 Components of a Data Parallel CEP system

An operator for Data Parallel Complex Event Processing can be comprised of the following components: a splitter consolidating the input streams and distributing them to the operator instances, which in turn match for the query, as well as a merger, that orders the stream of complex events from the operator instances [MTR17]. In this chapter we describe the tasks of the components in detail.

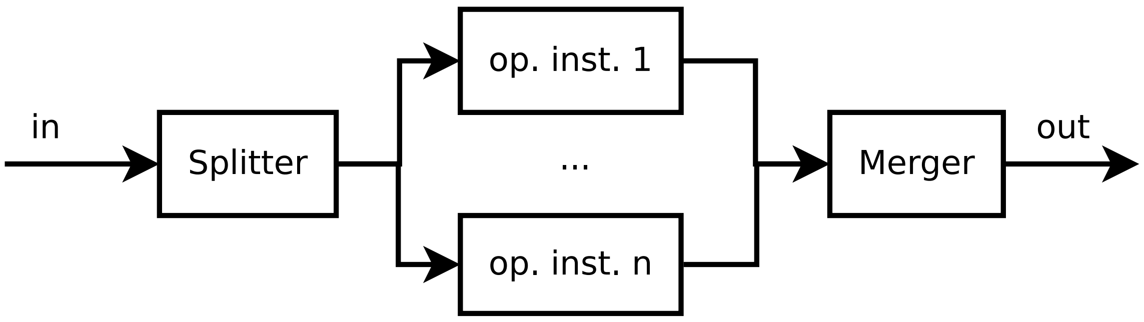


Figure 2.3: Typical setup for Data Parallel CEP, according to [MTR17]

Source

The source sends an ordered stream l of events to the operator ω. The source has an unique ID to identify the origin of an event as well as for ordering events with the same timestamps between streams. There can be multiple sources connected to a single operator. The number of sources is based on the topology of the system and is not specific to a Data Parallel CEP system compared to a classic CEP system without parallelization.

Splitter

The splitter is located in the operator and orders the event streams received from the sources $l_i \dots l_{i+n}$ based on their timestamps to maintain the total order of the events in the resulting, merged stream l_ω of events. For this merged stream, the aforementioned total order holds, such that

$\forall(e_i, e_j) \in l_\omega : i < j \Rightarrow e_i <_e e_j$

The splitter has to handle late arriving events. The lateness of events may be introduced for example by network latency. We assume that all events in l_i are ordered, so to assure that no late arriving event is lost, the splitter can wait until every source has provided an event, that has not yet been processed. This guarantees that no event is lost but can introduce severe latency, if one source pauses sending events. There exist other approaches, for example, it might be desirable to adhere to strict latency bounds, such that late events have to be dropped because they do not contribute to the current situating anymore [Hed17].

This merged stream gets split into windows w based on predicates P_{start} and P_{close}. These predicates are evaluated for every event in the merged stream to decide if a new window has to be opened or one or more existing ones have to be closed.

A scheduler within the splitter assigns created windows to connected instances and subsequent events in the window are sent to the assigned instance. Every window is assigned to exactly one instance, whereas one instance can process many opened windows. Since windows can overlap, the splitter has to send every event in the merged stream to all instances that process windows this event is assigned to.

Since typically many basic events create few abstract events, the eventrate on the splitter is much higher compared to the eventrate at the merger. Due to the high event rate and the fact, that all events flowing trough the operator have to pass the splitter, the performance of the splitter is critical for the overall system performance.

Furthermore, the number of instances the splitter can handle efficiently greatly affects the system scalability because it dictates to what extend the processing can be parallelized. A perfectly scalable splitter would support an infinite amount of instances without reducing its throughput.

Operator-Instances

An operator instance receives a single stream of events from its splitter and processes the events based on windows given by the splitter. If a complex event has been detected, the operator instance sends the complex event to the connected merger. The operator-instances are copies of the operator logic and can operate on different CPU cores or dedicated hosts. The number of operator instances limits the maximum achievable parallelization degree. The overlap of windows determines how many events have to be duplicated and sent to multiple instances.

The operator instances can process events in their windows on the fly as they arrive or wait until the whole windows was transmitted. This behavior can be configured or be required by the query. An example for a query that would require waiting is one that matches for the absence of events in a timeframe, in that case the operator instance has to wait until the window is closed by the splitter to be sure no event will arrive anymore.

The operator instances of a Data Parallel CEP system behave exactly the same as an operator in a classical CEP system, despite it being connected to a splitter and merger instead of being connected to sources and sinks or other operators.

Merger

The merger orders the streams of complex events originating from its operator instances to maintain a total order in the resulting merged stream of complex events. The merger provides this complex event stream, to whatever system is interested, for example another operator or a sink.

2.3.5 Sample scenario

To illustrate the connections between the components and their jobs, figure 2.4 shows a momentaneous state snapshot of an operator in a sample setup. The splitter merges l_1 and l_2 to an ordered stream of events l_ω. The events e_7 and e_8 have not yet been processed by the ordering algorithm but have arrived at the splitter, so it can further advance in processing the streams.

The predicates P_{start} and P_{close} are currently evaluated for e_4, which will not open or close any windows. e_5 will close w_2 after itself, while simultaneously opening w_3, in which e_5 will be the first event. e_3 is part of w_1 and w_2, which were scheduled on a different operator instance each and hence e_3 has to be sent to both operator instances.

Operator instance 1 has received e_1 and e_2 and will shortly receive e_3. Operator instance 1 is evaluating its query with the both already received elements but did not produce a match jet. The behavior of evaluating events of not yet closed windows, as they arrive, is optional.

Operator instance 2 has already produced a complex event c_1 that is being sent to the merger, that orders both streams of complex events. The complex event c_0 has already been recognized as the first complex event in the system and is sent out by the merger to further systems downstream.

Note that the depicted operator can be one of many others, so it can receive l_1 from a source or another operator, as well send the stream of complex events to an operator or a sink.

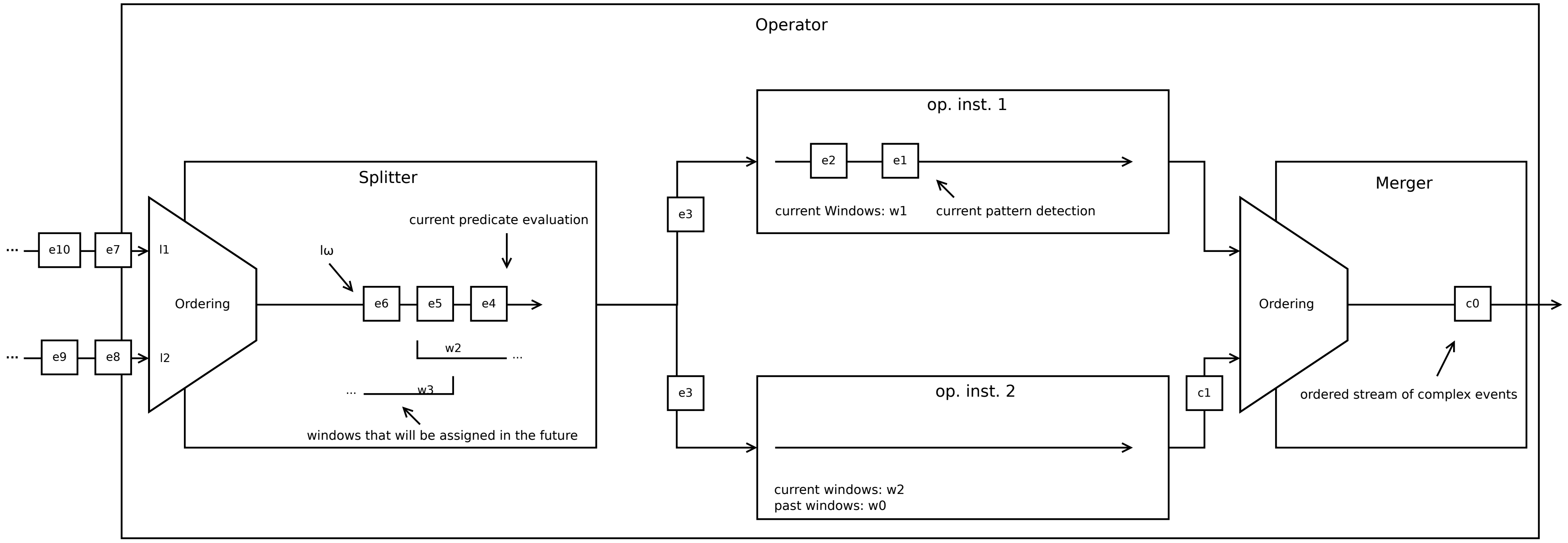


Figure 2.4: Sample setup of a Data Parallel CEP Operator

2.4 CEP frameworks

Complex Event Processing is a commonly used pattern [MST+17] and therefore there exist numerous frameworks to aid software developers with creating CEP applications [ZK15], [Gög18]. CEP frameworks offer the user different abstractions to work with events. The PACE framework, that is the base of our SPACE framework, is described in the next chapter. To later compare our SPACE framework with a commonly used framework, we present Apache Flink in this section.

2.4.1 Apache Flink

Apache Flink describes itself as "a framework and distributed processing engine for stateful computations over unbounded and bounded data streams"[Fou18b]. It is widely used in industry as well as academia [Fou18d]. Apache Flink allows to run CEP on streams passing through it, by providing APIs to its processing functions. Applications running on Flink can be distributed over multiple cluster instances managed by the framework.

The Flink client can submit work to a JobManager, that manages the execution of the work on multiple TaskManagers. These TaskManagers execute the actual work and communicate with each other if necessary. Since Apache Flink does not only support CEP, but also normal stream and batch processing, the framework incorporates a DataflowGraph [CKS+15], which can be compared to the operator graph used in CEP, in such that it defines how data is flowing between operators.

Flink allows to use several levels of its API to be used for CEP, the rather low-level DataStream-API as well as the higher level Pattern APIs. The DataStream-API allows to process elements with an arbitrary function, the so called `ProcessFunction` [Fou18e], an interface, whose implementation gets invoked for every event in a stream. Additionally, the `ProcessWindowFunction` allows processing on a whole window, as soon as it is closed. The pattern API [Fou18a] allows for designing patterns that can be detected on event streams, without the need to implement the detection logic as required in the low level APIs.

Apache Flink provides a lot of advanced features like tables, savepoints and distributed shared state [Fri16] that allow for more complex applications to be created but are not discussed in depth here.

Flink has several concepts of time, amongst them EventTime, that is solely based on timestamps extracted from or attached to events and ProcessingTime, representing the time of the operator, the moment it processes an event. To mark passing of EventTime, Flink introduces watermarks, that flow amongst normal events within event streams [Fri16]. Watermarks assure, that there are no events coming after them in the stream, that are older as the time noted on the watermark. Timestamps get produced by sources or are assigned based on the event contents by an assigner function upon ingestion. These two points of timestamp generation are also capable of generating watermarks.

Operator logic is executed in parallel within Flink by partitioning the stream of events. In contrast to the PACE and SPACE framework, Flink only supports partitioning by key [Fou18e], [Fou18b]. Flink supports partitioning the data stream into windows but cannot assign these windows onto multiple operators. The key-based partitioning approach is most suitable when dealing with unrelated data as described in Section 2.3.3.

Windows are created by a WindowAssigner, that handles opening and closing windows. There exist predefined assigners based on time or count but custom user logic is possible. Processing of a window can be based on a trigger, that defines when to start with processing based on certain events or time. Furthermore, there exists an eviction function, that allows to remove events from windows, even when windows are closed.

Flink can guarantee to keep the order of events in most cases, when working on one stream but cannot guarantee any order when a source has a parallelization degree > 1 or multiple streams get merged. The user of the framework is therefore required to provide an already ordered input stream. If there are multiple sources that need to be merged to one stream, an external program could supply a single merged stream of events in that case.

3 The PACE framework

In this chapter we give an overview of the analyzed PACE framework. First we explain its overall architecture and then detail on the components of the splitter and their tasks.

3.1 Architecture

To describe the PACE framework, we start by giving an architectural overview. The architecture is depicted in Figure 3.1 for reference.

3.1.1 Overview of components

The PACE framework supports one operator with multiple operator instances. The operator consists of the typical components of a Data Parallel CEP system, as described in Section 2.3.4, a splitter, instances and a merger. The Operator is connected to one or more sources, that directly connect to its splitter. The merger can provide complex events to external sources.

The framework is written in Java and runs on the Java Virtual Machine. The individual components are interconnected with TCP connections to guarantee delivery of events and prevent out of order delivery of events caused by the network.

The components are by default packed into separate jar files and executed from within these.

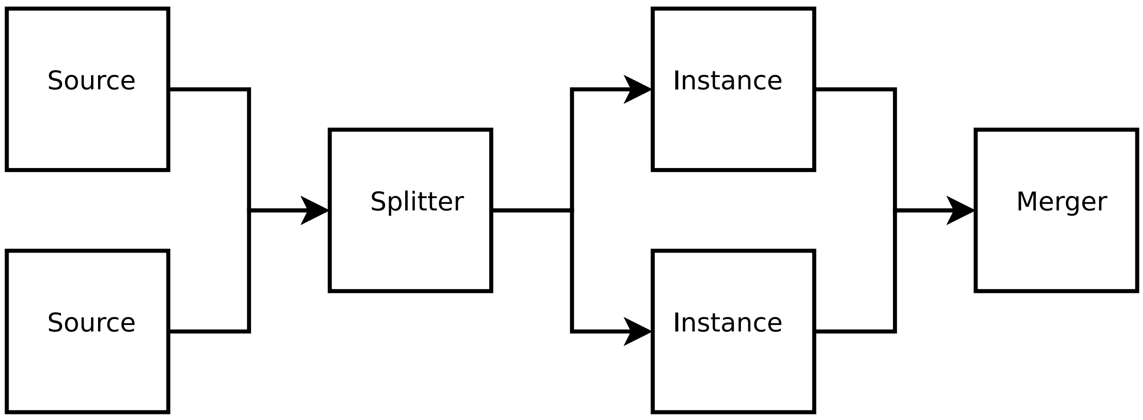


Figure 3.1: The components of the PACE framework

3.1.2 Source

The sources in PACE support multiple operation modes: generating events on the fly or reading already generated events from file. Every source has an identifier, that is unique for the system. Every source produces exactly one event stream with events having unique, increasing serial numbers. The events produced by sources have an attached timestamp, that is, depending on the operation mode, generated at event creation or read from file.

Generated events get marshalled into a transmittable format and get sent to the splitter for processing.

3.1.3 Splitter

The splitter consists of several components in separate threads that communicate via thread save queues. The components are described in detail in 3.2. The flow of events trough the splitter is as such: The SourceConnector receives events and passes them to the OrderComponent, after being ordered, the events are handed to the Manager, that takes care about windowing and distribution of events to finally be forwarded to the InstanceConnector, that sends the event to the instance.

3.1.4 Instance

The instance receives messages from the splitter, processes contained events and produces complex events if appropriate. Produced complex events get sent to the manager.

The instance keeps a map of opened windows and the according operators (implementations of the AbstractOperator class). Every event gets passed to the operator of every opened window for processing. Commands for opening and closing windows from the splitter add or remove entries from the map. The instance operators may buffer the events belonging to a window for processing or providing them to the complex event.

The operator logic can be defined by the users of the framework by providing an implementation of the AbstractOperator interface.

3.1.5 Merger

The merger collects complex events from the instances, orders them and provides them for other applications or users. Since we only use the merger in this thesis to validate the correctness of the execution, the merger writes its results to a file on disk.

3.2 Components of the splitter

The splitter is organized as depicted in Figure 3.2. In this chapter we describe the tasks of the individual components.

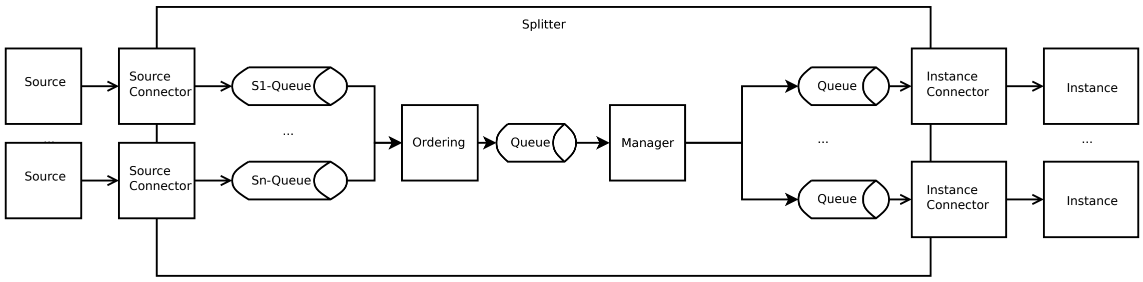


Figure 3.2: The components of the PACE splitter

3.2.1 SourceConnector

For every connected Source, the splitter creates a SourceConnector consisting of two threads, one for receiving messages and one for sending messages back to the source. Each thread has a stream connected to the socket, to which the source has connected. Messages that are received, are mainly events but the source connector is also capable of receiving control messages like requests for an unique identifier sent when a source initially connects. Messages that are sent back to the source can be, amongst others, messages reporting the current processing speed.

The SourceConnector unmarshalls received messages and adds them to a queue to provide them to the next component. Messages that need to be sent are likewise stored in a queue and are polled from there to be marshalled and sent.

3.2.2 OrderComponent

The OrderComponent orders the events from multiple incoming event streams. In the framework it is called 'Serializer' and is not to be confused with the Marshaller. Therefore, we decided to call it OrderComponent for simplicity.

This component is only needed if multiple sources are connected, if only one source is connected, the event stream already is ordered correctly. Nonetheless, PACE will always deploy a OrderComponent, regardless of the number of sources.

To illustrate the job of the OrderComponent, as described in 2.3.4, we provide a sample event sequence below. Assume there are two connected Sources, providing their streams l_1 and l_2. Let the indices of the events produced by these sources represent the timestamps (lower is earlier, so $e_1 <_e e_2$). Assume that the source of l_1 produced e_1, e_7, e_8 and the source of l_2 produced e_5, e_6 but the events arrive at arbitrary times at the splitter, caused for example by network latency. If we need to guarantee that no events are lost or out of order, the splitter needs to wait for e_1 and e_5 to arrive, since it cannot know if there are any events between e_1 and e_5. After e_5 has arrived, it can process e_1 and in turn has to wait until e_7 has arrived until it can process e_5 and e_6.

The details of the implemented algorithm for ordering the events are described in 4.3, as well as the proposed new algorithm for the SPACE framework.

3.2.3 Manager

The manager component of the splitter is responsible for evaluating the predicates P_{start} and P_{end}, to open and close windows on the event stream. Furthermore, it has to assign the windows to the corresponding instances, by calling a scheduler. Since an event can be part of multiple windows, it is possible that one event has to be sent to several instances. Every instance must receive every event only once.

The Manager in PACE keeps a list of opened windows as well as of connected instances. This list of opened windows gets updated with every positive evaluation of P_{start} and P_{end}. To support user defined scheduling, the list of instances is provided to the scheduler, to assign the window to an instance whenever P_{start} evaluates to true.

3.2.4 InstanceConnector

The InstanceConnector of the PACE-Framework resembles the SourceConnector, whereas the InstanceConnector provides two threads per connected Instance, one for sending and one for receiving messages. The communication patters are the same as with the SourceConenctor.

3.3 Method for analyzing the framework

In this section we describe the method we chose for analyzing the PACE framework to identify its shortcomings and define our goals for the SPACE framework.

3.3.1 Goals of improving the splitter

The overall goal of improving the splitter is to increase its throughput while keeping its functionality. Furthermore, it is necessary for the splitter, to be of use in real-world scenarios, to be able to maintain a constant event rate for an extended time period.

To make use of the Data Parallel CEP approach, the splitter has to be able to efficiently handle multiple instances.

To guarantee the stability of the splitter, the runtime of the system must not impact the maximum supported event rate. Otherwise the system cannot be seen as stable.

3.3.2 Measurements

We measure the throughput of the splitter by summing up the measured event ingestion rates at the instances. These rates get reported in short time intervals of 10 milliseconds to be able to visualize and detect changing behavior over time. If windows overlap and events have to be sent to multiple instances, the event duplication is taken into account. Additionally, the rates are computed as average rates over the whole runtime to detect measurement inaccuracies.

To investigate the stability, we fed the splitter with events for an extended time period and tracked its performance. Furthermore we monitored the resource consumption of the splitter throughout the process to assure it did not hog an increasing amount of resources.

3.3.3 Approach of analysis

To analyze the framework and detect bottlenecks that limit throughput, the splitter has been divided in distinct parts that were analyzed on their own to minimize side effects as far as possible. We analyzed these parts from the receiving side to the sending side of the splitter and incrementally improved them. The main parts of the splitter that we isolated are the SourceConnector, the OrderComponent, the Manager and the InstanceConnector.

The Individual parts were chosen based on the multi threaded nature of PACE, since each of these components has at least one own thread. To improve the parts of the splitter, we implemented periodical performance metrics to be able to analyze the behavior over time. We then assured that incorporating measures did not reduce the overall performance by comparing average results of benchmarks with and without measures.

As we now could analyze the behavior of the software, possible areas of improvement needed to be identified. We used several Java profilers to analyze aspects like garbage collection and CPU usage per thread. In addition to using the profiler, we analyzed the code by hand, made small modifications and verified our assumptions by running benchmarks to compare the performance.

The improved parts were then reassembled to the SPACE framework part by part, to be able to detect every change caused by side effects when reintegrating them. While changing subsets of the splitter code, benchmarks on the server cluster were conducted to assure the assumptions underlying the changes were correct and no other performance problems were introduced.

4 The SPACE framework

When deploying data parallel complex event processing, one of the main benefits is the possibility to distribute the operator instances onto multiple hosts and thereby distributing the work of detecting patterns in event streams. The PACE framework does not provide the desired performance properties and hence we cannot use it to efficiently distribute load onto multiple machines. Therefore we developed the SPACE framework (Speedy PArallel Complex Event processing framework) to solve the shortcomings of the PACE framework. In this chapter we describe our findings with the PACE framework and the changes applied to it, to form the SPACE framework.

4.1 Serialization

The components of the PACE framework communicate with messages of different types, e.g. events, windows and controlling messages. These messages reside in memory in form of Java objects. To send them over the network, we need to serialize them into a binary representation. In this chapter we describe how this serialization works in PACE and describe its drawbacks. We then present our proposed solution used in SPACE to overcome these problems.

4.1.1 Serialization in the PACE framework

In PACE, the messages are serialized to strings. The binary representation of these strings is then sent via the network. Upon receiving, the string gets deserialized back into a Java object. When profiling the application, we observed that this type of serialization and deserialization causes a lot of garbage string objects, that need to be collected by the Java Garbage Collector (GC).

With the profiler, we found that 59% of all garbage objects were created with the `String.split()` method while deserializing events. Furthermore, serializing events created another 31% of garbage objects. Of all garbage objects, 24% were `char[]` and 19% were strings.

This excessive object creation caused the GC to run more often and run longer and thereby reduced the performance of the system. Furthermore, we observed a filling heap space when running the system for a long time, what could be caused by a memory leak.

The deserialization mechanism works by splitting the received string based on tokens, generating an array of substrings. This imposes two drawbacks, first the deserializer has to traverse the whole string in order to match the tokens and second the generated substrings have to be removed by the GC, even if only a subset of the content is needed. The metadata values of an event, e.g. the arrival timestamp, are prefixed with a short string that identifies the following value. The event payload is saved as a HashMap per event in addition to be saved fields in the java object. This HashMap is serialized as key-value-pairs with human readable keys like `event_state`.

This way of serializing the events requires every event object to have a HashMap, that has to be filled, serialized and later removed, introducing further computational overhead. Furthermore, the keys identifying the values in the serialized event pose additional data that has to be processed and sent.

4.1.2 Serialization in the SPACE framework

Restructuring the event class In the PACE framework, every event is represented as a java object instantiating a class. This classes already have the event values implemented as fields, so it did not require additional work to serialize the java fields instead of the content of a HashMap. This allows us to remove the HashMap from the classes, making them more lightweight. Thereby the SPACE framework does not have to handle a HashMap per processed event, saving time when creating events when deserializing and causing less work for the GC. Since it is shown [NM02], that excessive object allocation reduces the JVM performance, we expected noticeable performance improvements.

Selection of a serialization library To provide a flexible and high-performant solution to serialize and deserialize events to be sent via the network, using external libraries is a promising approach [VK18]. Since Google protobuf (Protocol Buffers[1]) is amongst the fastest serialization and deserialization libraries available for Java, [PBYP17],[VK18] and the library is well maintained, it has been chosen to transform the Java objects of the events to a binary representation and back.

Protobuf is based on `.proto` files, which define the message layout. From these definitions, language specific code is generated, that provides containers holding the data. In the case of Java, classes with the message fields defined as class fields and means to serialize and deserialize these containers to binary data. Messages defined with protobuf can be nested, providing a powerful framework to model the communication of the PACE framework.

Layout of the message format The messages used in the framework can be put into three categories: Events, Windows and ControlMessages. Event messages contain the application payload, e.g. events representing stock market changes or events regarding the stream like an EndOfStreamEvent, that marks the depletion of a stream.

Window messages contain information about opened and closed windows. ControlMessages cover a wider spectrum of subtypes, for example a request for an unique identifier for a source at startup or feedback about the current throughput of a component.

[1]https://developers.google.com/protocol-buffers/

Based on the assumed message frequency, many events and windows but only few control messages, the following hierarchy has been defined to contain the messages.

The Message type contains common fields as the creation timestamp or the serialnumber of the message. This type has a payload field containing either a WindowMessage, an EventMessage or a ControlMessage. The multiple types of the payload field are based on the `oneof` type in protobuf, resembling an enum known from several programming languages.

To identify the type of the `oneof` field in protobuf, an index field in the message defines the position of the type in the declaration of the message. Since the index of the subtypes (see Listing 4.1) is known to the serialization and deserialization components, the system has to read only the index field to detect the type of the contained payload field.

The types that can be put into the payload field of our `Message` definition are containers encapsulating the real message-payload. This layout has been chosen as a trade-off between maintainability and performance, as described in section 4.1.2.

Listing 4.1 The protobuf declaration of the Message

```
message Message {
  int64 sn = 1;                // serialnumber of this message for the stream it appears in
  int64 timestamp = 2;            // timestamp of the message
  int64 originId= 3;            // unique identifier of the origin component
  oneof payload {              // a message payload is either an Event, Window or a ControlMessage
    EventMessage evt = 4;
    ControlMessage ctl = 5;
    WindowMessage wnd = 6;
  }
}
```

The EventMessage acts as a container for the actual type of events that have to be sent. It contains another `oneof` field explicitly declaring the possible types of the payload field, for example EndOfStreamEvent or SimpleEvent. These payload types contain the events that should be processed, as an example the content of the `StockEvent` definition, representing changes in the stock market, is shown in 4.2.

Listing 4.2 The protobuf declaration of the StockEvent

```
message StockEvent {
  string symbol = 1;
  float open = 2;
  float high = 3;
  float low = 4;
  float close = 5;
}
```

The WindowMessage has the same structure as the EventMessage except that it contains window subtypes instead of event subtypes.

The ControlMessage consists of an `any` field, that contains the payload of this message. This field type can contain any other encoded protobuf message. This option has been chosen because there are many types of ControlMessages, for example ConnectRequests or acknowledgements, that are likely to be expanded in future versions of the framework, making explicitly declaring every possible type not very maintainable.

Maintainability and Performance of nested messages in protobuf The two methods used for nesting messages, using `oneof` and `any` fields, have both benefits and drawbacks, that have to be considered.

The `oneof` type requires the author of the software to explicitly declare every possible type the field can contain. This enables the framework to encode the type as an index in the message making serialization and deserialization efficient at the cost of more maintenance work, when these types change.

In contrast to this approach, the `any` field can contain any other message and therefore has to encode the contained type in an unique way. This is realized by encoding the name of a type as a string in the message, for example `type.googleapis.com/messages.ConnectRequest`. This method of nesting messages enables easier changes in the framework at the cost of lower performance, because the type string has to be sent with every message and when deserializing the message, the type string has to be explicitly checked for every message.

So with this trade-off in mind, we chose the `oneof` approach for performance critical messages with high throughput like events and windows and the `any` approach for the more flexible messages of lower frequency like the ControlMessages.

4.2 Networking

The PACE framework is built upon BufferedReaders on top of Java TCP socket connections between the components. Messages get sent as strings, delimited by line endings, and are processed on the receiving side. In this section we describe the network implementation of the PACE framework and describe its drawbacks. Then we detail how these problems were approached in the SPACE framework.

4.2.1 Networking in the PACE framework

To ensure that the messages are sent with low latency, the `flush` method is used after every message. Flushing the DataOutputStream forces the underlying stream to flush, in this case the SocketOutputStream. This causes the JVM on Linux to call `send()` of `sys/socket.h`[2]. Since the serialized representation of a SimpleEvent has a size of 63 bytes, the system has to make a syscall every 63 bytes of payload for the SimpleEvents, negatively impacting the performance.

[2]see https://github.com/unofficial-openjdk/openjdk/blob/68a83a60bbed038c5872ccfb8be0ade0ca271507/src/java.base/linux/native/libnet/linux_close.c#L386

The PACE framework starts a receiver and sender thread in the splitter for each connected instance and source. This approach allows for an easy implementation but requires the system to manage a lot of threads. The large amount of active threads negatively impacts performance because it requires a lot of context switching in the JVM. Therefore this approach limits scalability.

The PACE framework communicates with a line delimited, string based protocol. The line based approach of the original PACE framework partitions the TCP stream into discrete messages by appending a line ending to every message. The receiver can then process the continuous TCP stream by splitting it into strings on received newlines. This tokenization does not work for protobuf, since protobuf is a binary protocol and does not honor the newline as a special, reserved character.

In regard of the high efforts needed to implement message framing for protobuf with the Java Socket-API, the large amount of syscalls and the threading model, we decided to rely on an external networking library.

4.2.2 Selection of a networking framework

The networking framework needs to be able to handle protobuf messages with framing to provide a message abstraction and be able to handle lots of events, consisting of many small messages, in an efficient manner. ZeroMQ has been chosen for this task because it fits these requirements well. ZeroMQ is based on a Message abstraction and implements its own lightweight framing for these messages. Furthermore ZeroMQ is proven to be amongst the fastest networking libraries [PBYP17] available for Java.

4.2.3 ZeroMQ based Architecture

ZeroMQ[3] provides an abstraction called Socket, that differs the paradigm underlying the Java Socket-API. A ZeroMQ-Socket can be connected to multiple endpoints and can have one of multiple types, details are explained in the ZeroMQ-Documentation[4]. The ZeroMQ sockets can be muted, to stop sending messages, this leads either to discarding them or to block. Every connected socket has an ID to be identified by its counterpart. The socket types relevant for the framework are explained below.

- DEALER: The DEALER socket allows for pushing messages to another connected socket and blocks if it is muted. This socket type also allows for receiving messages from connected sockets. When sending messages, the DEALER socket selects the endpoint according to a round-robin-scheme, received messages are fair-queued.
- ROUTER: The ROUTER socket fair-queues receiving messages from all connected endpoints and prepends their ID to the messages for later identification. The ROUTER socket is able to send a message to an endpoint specified by its ID.

[3] http://zeromq.org/

[4] http://api.zeromq.org/4-2:zmq-socket

- PUSH: The PUSH socket distributes messages equally to all connected peers. It will block if it is muted.
- PULL: The PULL socket fair queues received messages amongst all connected peers and adds them to one internal queue.

Messages received by a socket are queued internally and can be polled in a blocking or non-blocking manner. Messages sent to a socket are also queued internally until they can be sent.

The following communication layout is implemented for SPACE:

Source-Splitter We modeled the connections between sources and the splitter as a single PULL socket in the splitter and a PUSH socket in each source. This allows the framework to have a single receiver thread, that covers all connected sources and guarantees fair-queuing amongst them. Despite bidirectional communication is possible, it is only used for an initial handshake, subsequent messages are sent only from the source to the splitter to have a cleaner architecture in the splitter.

Messages that have to be sent back are handled via the FeedbackSocket, described in detail in the next paragraph.

Since the identifier of the source is contained in every event, the PULL socket does not need to know the identity of the other socket to be able to identify the event origin.

Splitter-Instance Since the splitter has to assign windows to instances, it has to be able to send an event to specific instances. The ROUTER socket allows the framework to achieve this by mapping the internal instance number used for assigning windows to the identifier of the connected ROUTER socket of the instance.

FeedbackSocket For an architecture, that is easier to understand and maintain, communication opposing the flow direction of events, namely from the instance to the splitter and from the splitter to the source, has been extracted into the FeedbackSocket. This is realized as a ZeroMQ ROUTER socket, to which every Source and Instance connects via a DEALER socket. This enables decoupling of communication regarding events and other communication. Furthermore it provides an easier to maintain architecture because every socket, except the FeedbackSocket, has to communicate in only one direction.

4.3 Ordering algorithm

The Ordering component of the PACE framework is responsible for maintaining a total order of events in its output stream, that consists of the combined input streams.

4.3.1 Underlying assumptions

The design of the ordering algorithm is based on the following assumptions:

Total order There exists a total order of events. This is implemented as follows: first order by timestamps, then order by sequence number within the originating event stream, then order by the identifier of the input event stream. With this order mechanism, it is assured, that there exists a total order between the events of different input streams, that get merged into a single output stream.

Ordered input streams The ordering algorithm assumes that the events it receives in one input stream already are ordered. This is guaranteed by using TCP as transport between components and queues between threads, if the events are ordered at the source. Therefore the implementation of the source has to make sure it provides its events in an ordered manner.

Sources are equally fast Since we must maintain the order between streams of events that can arrive at arbitrary times at the splitter, the ordering has to wait for every input stream to have at least one event available before being able to process any other event. This situation is explained in more detail in section 2.2.1.

Number of Sources It is assumed that the number of sources per splitter is kept as low as possible.

4.3.2 The original ordering algorithm

PACE uses a binary tree to sort the incoming events. First an event from every input stream in a list of streams to poll events from is added to the tree. Then the stream is removed from the list so it is not checked again, until the already extracted event is processed. Then the next-to-be processed event is determined in the tree, removed from it, processed and the corresponding input stream is again added to the list of input streams to poll events from. The following pseudocode in Listing 4.3 illustrates this algorithm.

4.3.3 Problems arising from the original ordering algorithm

The PACE ordering algorithm has to create and delete a list node and a tree node for every processed event. This produces excessive amounts of objects that have to be removed by the GC.

We implemented a different algorithm that produces less garbage objects to handle this problem.

4.3.4 The SPACE ordering algorithm

We designed the new ordering algorithm to keep allocated objects to a minimum to reduce the amount of objects the Garbage Collector has to delete. This requirement rendered the approach of using a TreeMap to sort the events efficiently infeasible. Also, the approach of keeping a list of streams, that have to be checked for new events, by adding and removing the streams when an event was processed, produces more garbage objects than absolutely necessary.

The new ordering algorithm introduces a new data structure, the ReadHead, which represents the end of a stream of events facing the splitter. The events in the stream have not yet been processed. The ReadHead has an additional field containing the last unprocessed event, the event that has been polled from the stream for comparison with others but has not yet been processed and passed to the manager.

The idea of the new algorithm is as follows: The splitter has a list of connected source streams represented by ReadHeads. It has to decide which element to process next based on the order of events. The next-to-process event e_a is the one, that is the smallest compared to all other unprocessed events. The splitter has to determine the ReadHead H_a containing e_a as well as the ReadHead H_b containing the second smallest event e_b. The splitter can then process e_a and all events in H_a until the next event in H_a is larger than e_b. This event gets saved as the new next-to-process event of its originating ReadHead and the splitter has again to determine the ReadHeads with the smallest and second smallest events. The simplified version of this algorithm is shown in Listing 4.4.

Since the algorithm has to wait for an event to be available on every source stream and these source streams are ordered, it will always process the smallest available event and thereby maintain the total order of events in its output stream. The runtime of the implementation of `smallest_in` and `second_smallest_in` is linear to the number of sources but does not create garbage objects. This allows the splitter to process streams at higher rates, as long as the number of sources is reasonably low.

Listing 4.3 Original Ordering algorithm

```
streams_to_check=[]
tree = TreeMap()
for source in sources:
  streams_to_check.add(source.stream)
while running:
  for stream in streams_to_check: //check all unchecked streams before processing
    streams_to_check.remove(stream)
    stream.receive(event) //blocking
    tree.add(event)
  event = tree.take_first()
  process(event)
  streams_to_check.add(event.origin_stream)
```

Further challenges The ordering algorithm has to wait for an event to be available from every source to continue processing. This can lead to large latencies if the connected sources do not produce events at equal rates. An example for such a case could be a CCTV camera, that has to monitor a railway crossing, where the opening and closing of the barriers as well as the video frames are events. As soon as the close-event of the barrier has been registered and processed, the ordering would have to wait until another event from the barriers arrives, for example the open-event. In the meantime the camera frames cannot be processed as events. To handle this kind of scenario, it would be possible to implement heartbeat events that are sent by a source to notify the splitter that there have been no events in the meantime and then can be dropped.

This approach could be implemented in future versions of the framework.

4.4 Manager algorithm

The Manager component of the CEP framework is provided with one stream of ordered events by the OrderComponent and has to decide if an event opens or closes windows and to which instances the event has to be sent. This is achieved by calling the open and close operations of the user-defined `Operator` classes as well as by incorporating calls to a `Scheduler` class. The scheduler decides to which instance a window is assigned. Since an event can belong to multiple windows, it has then to be sent to all instances that currently have opened windows. The simplified pseudocode is shown in Listing 4.5.

Listing 4.4 SPACE Ordering algorithm

```
heads[]
for source in sources:
  head= new ReadHead of source
  first = head.poll()
  head.unprocessed=first
  heads.add(head)
while running:
  smallest = smallest_in(heads)
  next = second_smallest_in(heads)
  do:
    process(smallest.unprocessed)
    smallest.unprocessed=smallest.poll()
  while(smallest.unprocessed < next.unprocessed)
```

Listing 4.5 Simplified PACE manager algorithm

```
listOfWindows[]
method process(event){
  if predicateStart(event):
    window = new Window()
    listOfWindows.add(window)
    instance = schedule(window)
    send(instance,window)

  closedWindows[]
  for window in listOfWindows:
    event.assignedInstaces.add(window.instance)
    if predicateClose(event,window):
      listOfWindows.remove(window)
      closedWindows.add(window)

  for instance in event.assignedInstaces:
    send(instance,event)

  for window in closedWindows:
    send(window.instance,window)
}
```

4.4.1 Problems arising from the original manager algorithm

By profiling the application, we observed that the manager produces a lot of garbage objects. Furthermore, the manager algorithm contained several lookups with linear runtime, that could be modified to have constant runtime.

To tackle this problems, we modified the algorithm in the following ways:

- Instead of keeping a list of instances an event is assigned to, the event is directly sent to the respective instances. This removes the need to allocate a list, add elements, iterate over it and then remove the list and the items with the GC.
- Instead of keeping a list of closed windows per iteration, the windows are directly closed and the information about the closed window is directly sent to the instance.
- Since the instances are connected with ZeroMQ to the splitter, it is now possible to address an instance directly by its instanceID instead of a dedicated InstanceConnector. This removed the need of the `getInstanceConnector` method (not shown in the pseudocode) which has to iterate over all InstanceConnectors to determine the correct one, thereby wasting resources. This is now replaced by a lookup of the identity of the ZeroMQ socket with constant time complexity.

4.5 Rate control

We added a RateControl to the SPACE framework, so the splitter can control the event rate of the sources. This is useful to prevent the source to send more events than the splitter can handle.

4.5.1 Overload protection

The RateControl prevents a buildup in the number of events stored, caused by a slow OrderComponent or Manager, so the GC has less work to do.

Furthermore, this prevents the network related buffers to be constantly full, because the next component in the processing chain can't keep up with the event rate.

The RateControl on the splitter is implemented by summing up the current average processing speeds of the instances, received by the splitter, comparing them to the desired speed of the sources and generating a new desired rate based on certain conditions like a maximum deviation. Furthermore the splitter knows the total amount of events it has buffered or are being processed. With that information, the splitter can decide if it will be overloaded soon, for example if the number of buffered events grows significantly.

4.5.2 Warm-up phase

The RateControl further provides the possibility to gradually increase the rate of events at system startup. In this warm-up phase, the system is significantly slower. If the sources push at the maximum supported rate of the system already in this phase, the system cannot cope with the amount of events and will buffer them, leading to performance issues.

If the event rate is gradually increased at system startup, higher maximum rates can be achieved.

4.6 Architecture of the SPACE splitter

Figure 4.1 depicts the flow of messages in the SPACE framework. To distinguish control and feedback messages from events, their opposing flows are highlighted in different colors. The Instances and the Sources are depicted to clarify the connections to them but are not detailed further.

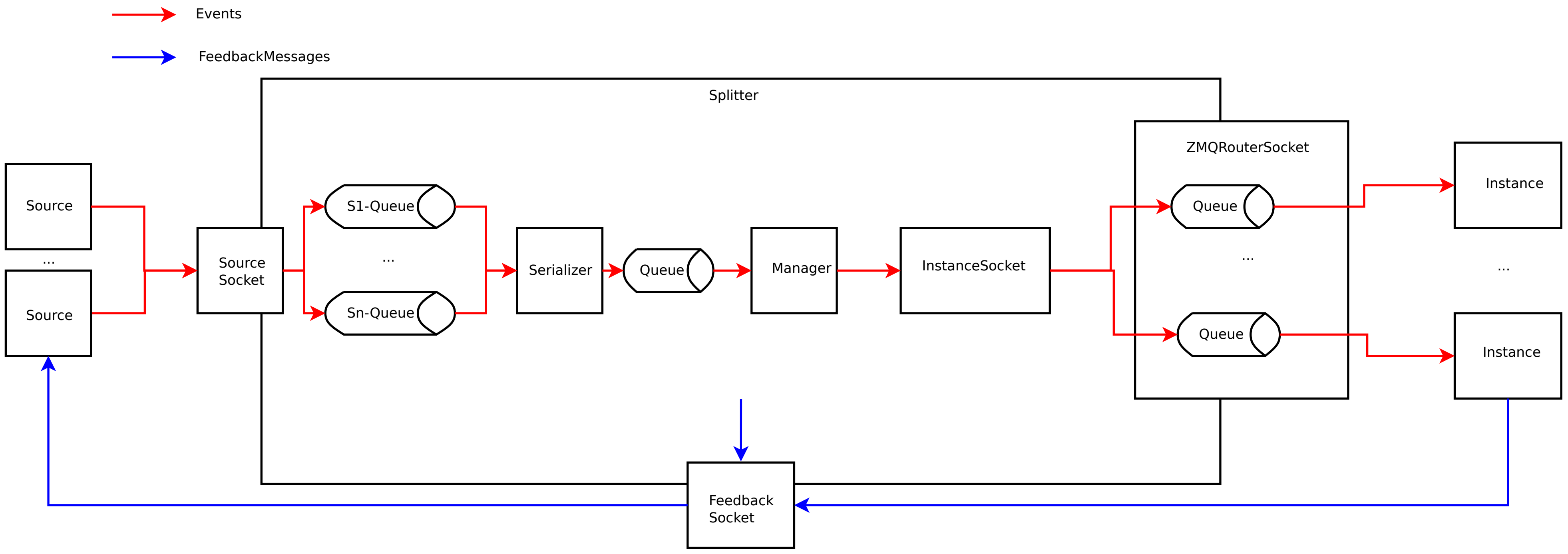


Figure 4.1: The message flow of the SPACE framework

5 Results

To evaluate the improvements described in the previous chapter and to compare the capabilities of SPACE with an industry proven framework, we conducted several benchmarks. In this chapter we outline the procedure and results of our benchmarks and discuss our findings.

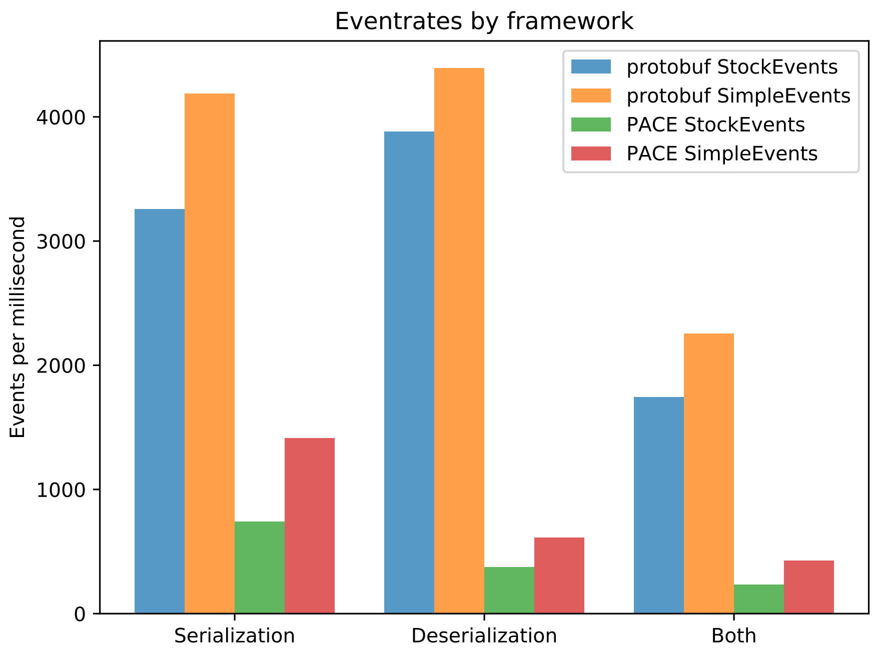
All benchmarks were run on a cluster of 7 servers interconnected via gigabit connections. All of these servers are equipped with Xeon E5620 CPUs with 12MB cache and are running at 2.4GHz. Of the available servers, 6 have 8 CPU cores and 24 GB RAM and one has 16 CPU-cores and 24 GB RAM available. We ran the benchmarks on Java 1.8.0 HotSpot 64-Bit Server JVMs.

5.1 Comparison of the serialization performance of the SPACE and PACE frameworks

To evaluate the performance impact of the changed serialization approach, we conducted a benchmark using the two technologies.

We evaluated both approaches by measuring the rate of serialization, deserialization and both, first serializing and then deserializing events. This has been tested with the SimpleEvents, only containing metadata and a single character of payload, as well as StockEvents, containing the same amount of metadata, five floats as well as a three character identifier.

The sizes of the serialized events are depicted in figure 5.1b, showing a considerable overhead when using the PACE serialization.



(a) Rates of event processing

	StockEvent	SimpleEvent
Protobuf	38	16
PACE	155	63

(b) Size of serialized events in bytes

Figure 5.1: Results of the benchmarks

The results of the benchmarks are shown in figure 5.1a. We noticed a significant performance improvement when using protobuf instead of the PACE serialization. The deserialization performance of protobuf is about 10 times higher than the PACE deserialization performance. By using the profiler, we could observe that the excessive garbage object creation mentioned in 4.1.1 was gone.

Furthermore, the PACE serialization showed a larger percentual change when increasing event complexity, possibly indicating even further performance problems when the events have more fields that have to be serialized and deserialized.

5.2 Maximum throughput of the splitter

We conducted benchmarks to measure the maximum event rate supported by the splitter pipeline. Since the splitter has the least work to do if there is only one connected source and one connected instance, we chose the following scenario for this benchmark.

5.2.1 Experimental setup

Each framework was evaluated with the same settings as described in this section. One sufficiently fast source is connected to the splitter, which in turn is connected to a single, sufficiently fast instance. We measured the event rate of the splitter as the rate of events received by the instance.

We assured that the arbitrary chosen window size (1000 events) did not impact the framework performance.

To rule out unfair optimizations of the dataset, we tested with the trivial SimpleEvent and the StockEvents. The SimpleEvents contain only a single character as their payload, the Stock event contain five floats and a three character identifier as their payload. The presented results are averages of 5 repetitions each. The instances in this scenario discarded the events upon receiving so they would not negatively impact the splitter performance by posing a bottleneck themselves.

5.2.2 Results

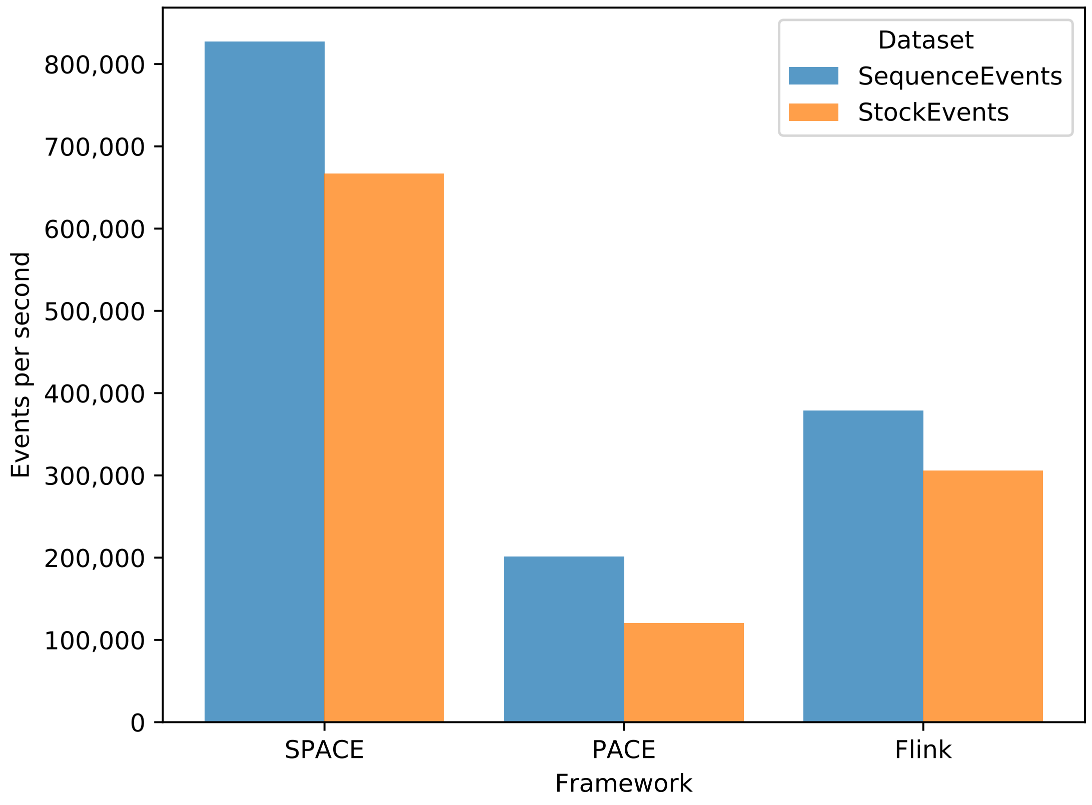


Figure 5.2: Throughput of the splitter of different frameworks

The results of the benchmarks are depicted in figure 5.2. When conducting the benchmarks, we could observe a speedup of 553% of SPACE compared to PACE. Since this is only the maximum supported event rate of the splitter, we further describe the behavior of these two systems in the next few chapters.

Furthermore, we observed that the SPACE framework outperformed Apache Flink in this benchmark. A possible reason for this could be the specialization of the SPACE framework. Apache Flink is capable of Batch-Processing, Stream-Processing and Complex Event Processing and therefore has to provide a more general approach. In addition, it supports many advanced features like distributed snapshots and watermarking, that, despite not being used in this benchmark, require a more complex architecture. In contrast to this, SPACE is a purpose-built, custom framework.

5.3 Window overlap

If patterns on a stream cannot be processed on windows that do not share events, the event detection system has to support overlapping windows. To evaluate the performance of the frameworks when using overlapping windows, we conducted the benchmarks described in this chapter.

5.3.1 Experimental setup

We want to know how well the PACE and the SPACE frameworks perform with a lot of opened windows. Both systems save information about currently opened windows in their splitter to send events to the instances, the windows are scheduled on. To evaluate the performance, we ran benchmarks with different amounts of opened windows. We chose a window size of one million events and changed the overlap to get the splitter to keep a number of windows opened. We then measured the maximum event rate with that configuration. In this experiment, we did not duplicate events but only served one instance.

5.3.2 Results

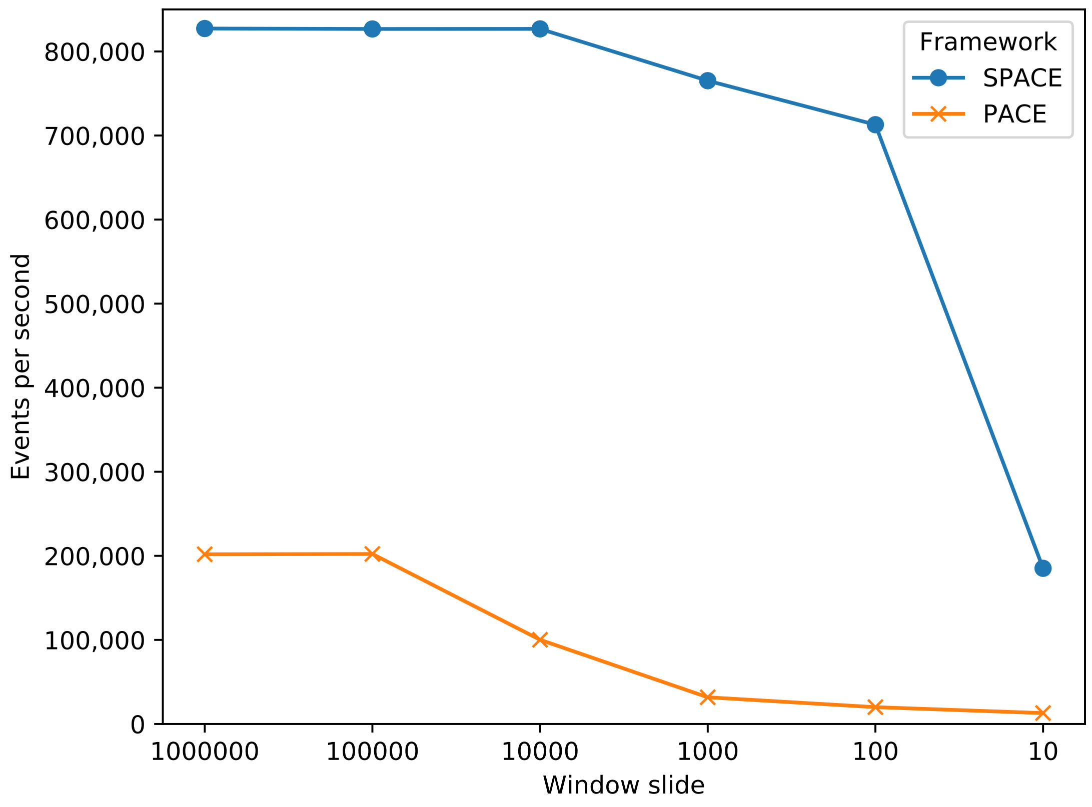


Figure 5.3: Throughput of the SPACE and PACE frameworks with different window slides.

Figure 5.3 shows the event rates measured. Note that the x axis is not linear. We observed that the SPACE framework can support more opened windows compared to the PACE framework, without a significant decrease in performance.

However, with a window slide of 10, what equals 10^5 opened windows, the performance decreases severely. This might be due to the checking of the predicates in the splitter, that has to be done for every opened window.

We conclude from this experiment that the SPACE framework is capable of handling more opened windows than the PACE framework, but will show decreased performance if the amount of windows is too high.

5.3.3 Stability

The PACE framework has shown unstable performance when running for some time in the past. To reproduce this behavior, we slightly overloaded the splitter and measured its performance over time. The eventrate is shown in Figure 5.4, the moving average is depicted in orange for simplicity.

We observed a clear decline in performance after a few seconds.

The implemented RateControl in SPACE is partially able to prevent such scenarios, depending on the speed of the change-rate of the event rate. The RateControl handles this by adjusting the event production rate. If such a change is not feasible, other, more drastic measures like ignoring events, have to be taken. Examples for working and not working scenarios are shown in Figure 5.10: In the experiment '250 2' it did not work, in '500 2' it worked.

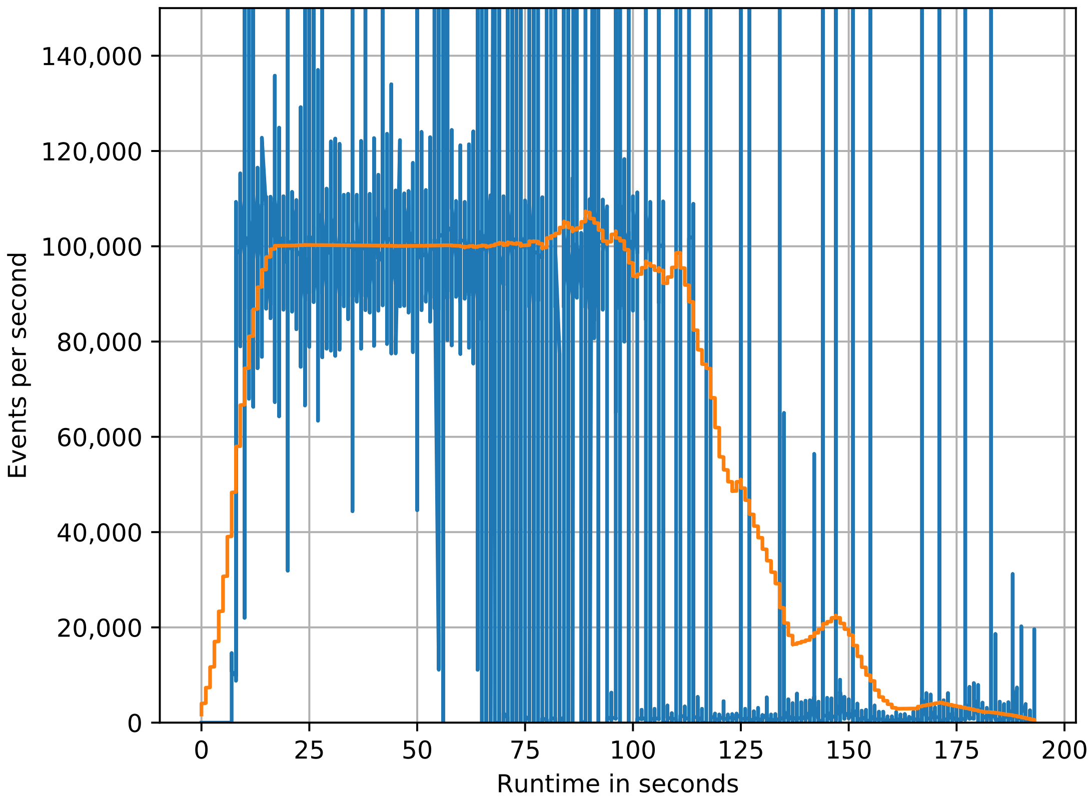


Figure 5.4: Event rate over time

5.4 Scalability

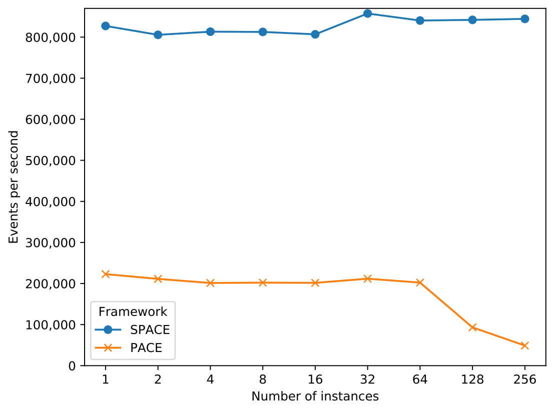
An efficient splitter has to be able to support many instances to be able to distribute the work on multiple CPU cores or computers. Therefore, we conducted several benchmarks to investigate the scalability of the frameworks as described in this chapter.

To give an overview of the experiments, they are listed briefly below:

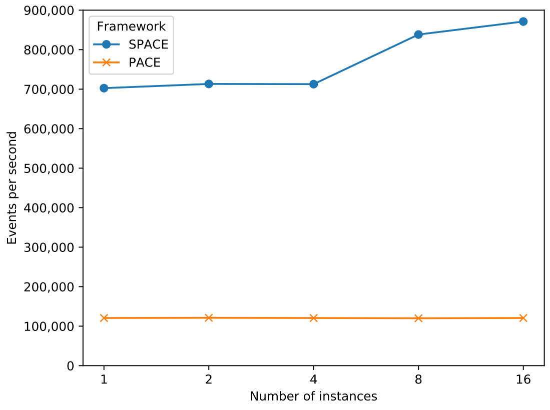
- Multiple instances without window overlap
- Multiple instances with window overlap
- Multiple sources
- Slow instances

5.4.1 Instances

To investigate how many instances are supported by the frameworks, we ran them with the SimpleEvents and StockEvents without any overlap and a changing number of instances. We also conducted a benchmark with windows that overlap, that one is described in chapter 5.4.3.



(a) Using SimpleEvents



(b) Using StockEvents

Figure 5.5: Maximum supported rates with a different number of instances

In Figure 5.5a we present the event rates of the frameworks with the SimpleEvents and the variable number of instances, also note, that the x scales are not linear. We could observe no negative impact when increasing the number of instances for the SPACE framework, if the windows do not overlap. The PACE framework however, showed decreased performance when the number of instances was larger than 64.

Figure 5.5b depicts the results of the same benchmarks using StockEvents. The PACE framework does not show any negative impact when the number of instances is increased. This contrasts with the observations when using simple events. One possible explanation for this phenomenon could be that the more complex StockEvents pose more work for serializing, deserializing, sending and receiving and hence the overall throughput might be limited by these operations and not necessarily by the manager algorithm, as suspected when using SimpleEvents.

In the case of the SPACE framework with SimpleEvents and StockEvents, we observed an increase of the event rate when the number of instances increased. This effect was more distinct, when the system was running with StockEvents. This effect needs more investigation, since we could not clearly identify the cause of it in this thesis.

We assured that the following reasons were not the root cause for this effect:

- A slow instance could have caused a similar effect. We measured the speed of the instances used in the scenario and noticed a higher throughput than the splitter could provide, so the instances seem not to be the bottleneck in this case.
- Insufficient network speed at some nodes could cause a similar pattern. To rule out this possibility, we measured the available bandwidth between all nodes and found it to be greater than the bandwidth needed by the application.
- Inaccurate measuring could be ruled out by measuring in different ways, comparing the results and running the experiment multiple times.

5.4.2 Sources

One of the main tasks of the splitter is maintaining the total order of events originating from multiple input streams. To evaluate the framework performance at this task, we ran benchmarks with a sufficiently fast instance and several sources and measured the event rate.

Apache Flink does not support ordering events between multiple streams natively, therefore we excluded it from this benchmark.

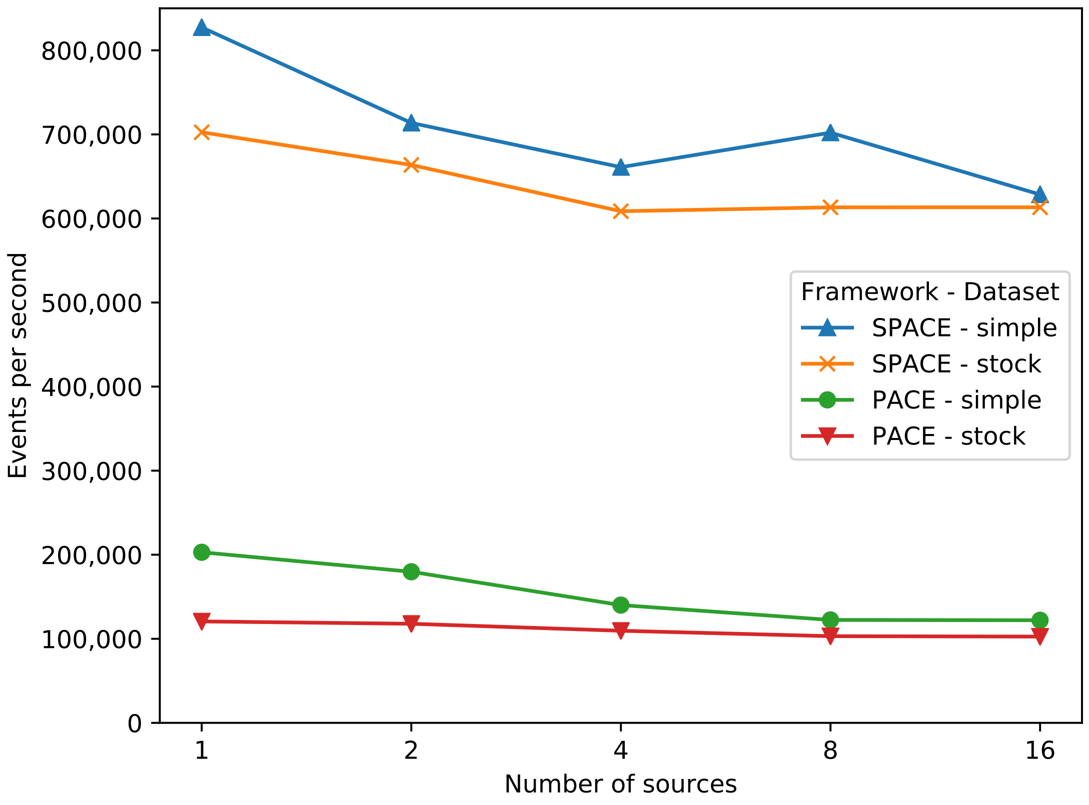


Figure 5.6: Throughput of the SPACE and PACE frameworks for different numbers of sources

Figure 5.6 depicts the results of the benchmarks. We observed decreasing event rates with increasing sources for the SimpleEvents for the PACE and the SPACE framework. It is notable that the comparably high event rate of the SPACE framework with a single source is due to a shortcut that circumvents the ordering algorithm if only one source is connected.

Furthermore, it is notable that the event rate of the PACE framework does not decrease that much, if it is running with StockEvents (-14% for StockEvents compared to -66% for SimpleEvents when increasing from 1 Source to 16 sources). This suggests the following explanation: The PACE-splitter is bottlenecked by the serialization and deserialization component instead of the ordering component when the larger and more complex Stock-Events are used. Therefore, the effect of the increased work of the ordering component might be hidden.

5.4.3 Event duplication

To efficiently handle windows that overlap, that are scheduled on multiple instances, the splitter has to be able to duplicate events in a performant manner. To investigate the behavior of the SPACE framework in this regard, we conducted the following experiment.

When the splitter is configured to work with sliding windows with a window-size of 20 and the window slide is 10, there are constantly two opened windows. If there are two connected instances, the events have to be sent to both instances, if we assume the windows are scheduled in a round-robin fashion. If the window slide is 5, there are four windows opened at any time. When there are two instances connected, the splitter has to send twice the amount of event it receives, as in the preceding scenario, if there are four instances, it has to send four times the amount of events it receives, because at any time, there is an active window on any instance.

In this benchmark, we ran the SPACE framework with a constant window size (one million events) and different window slides and a variable number of instances with SimpleEvents.

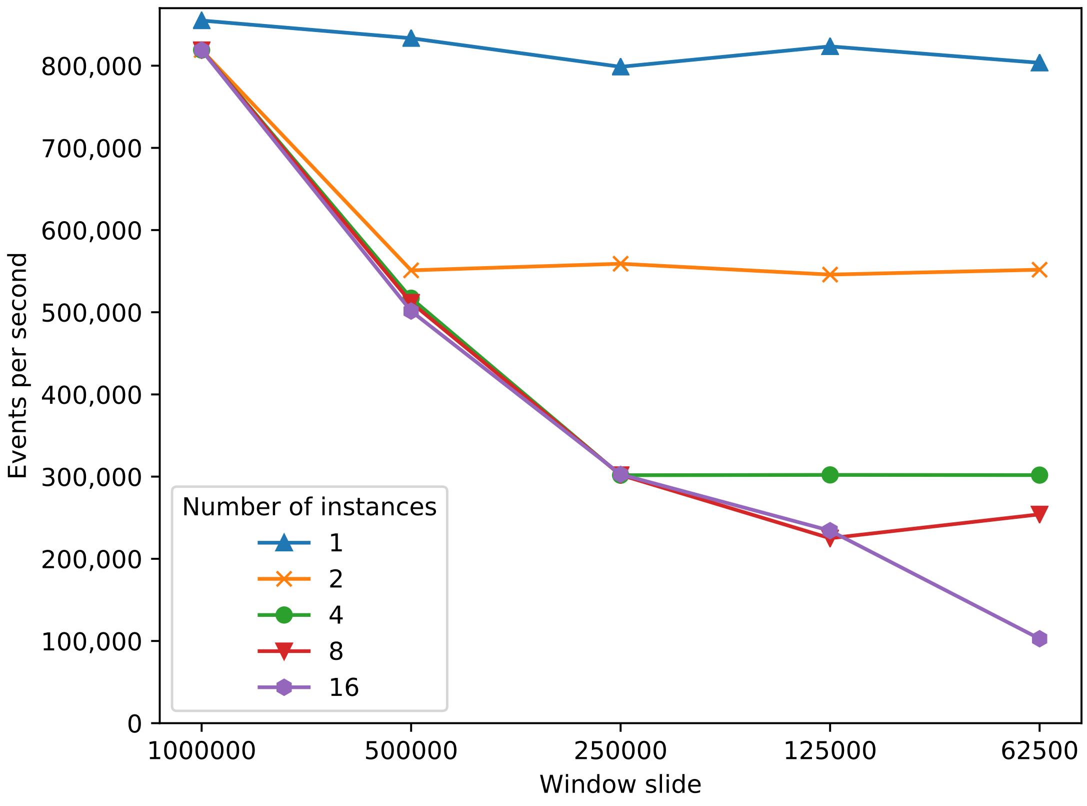


Figure 5.7: Throughput of the SPACE framework with different window slides

Figure 5.7 depicts the result of the described benchmark. Note that the x axis is not linear. We observed that the duplication of events impacts the splitter performance. It is clearly visible that if there are more opened windows than instances, the performance won't decrease further. We also observed that if there are less windows than instances, there is no real overhead compared to equal amounts of windows and instances.

We further get the maximum supported ratings for a duplication factor of 2 and 4 for the next benchmark.

5.4.4 Busy Instances

All experiments above were run with instances that were able to consume more events than the splitter could send. Since the instances need to be slower than the splitter for the parallelization to pay out, we simulated the case of slower instances in this experiment.

In the following three experiments, we ran the three frameworks with instances that can process 35000 events per second. We adjusted the event production rate on the sources so that it equals the amount of events processed by all instances. With this setup, we wanted to investigate to which extent the splitter can scale without imposing the bottleneck in a real-world scenario. We ran this experiment with a window size of 1000 and window slides of 500 and 250.

We refrained from determining the optimal speed for the sources for every configuration since we wanted to observe how long the systems scale proportionally, how they behave when being overloaded and because it would have taken a significant amount of time to do so. Since Apache Flink does not support the same features as SPACE and PACE, we chose to use the fastest possible solution, that does roughly the same as the other solutions.

Apache Flink Figure 5.8 depicts the eventrate for Apache Flink in this experiment. We observed Flink to adjust the speed of the source to its needs, therefore it did not break down in performance. Flink did not scale directly proportional to the number of added instances, nonetheless it did scale nearly linearly to its maximum rate at roughly 330000 events per second and then stagnates.

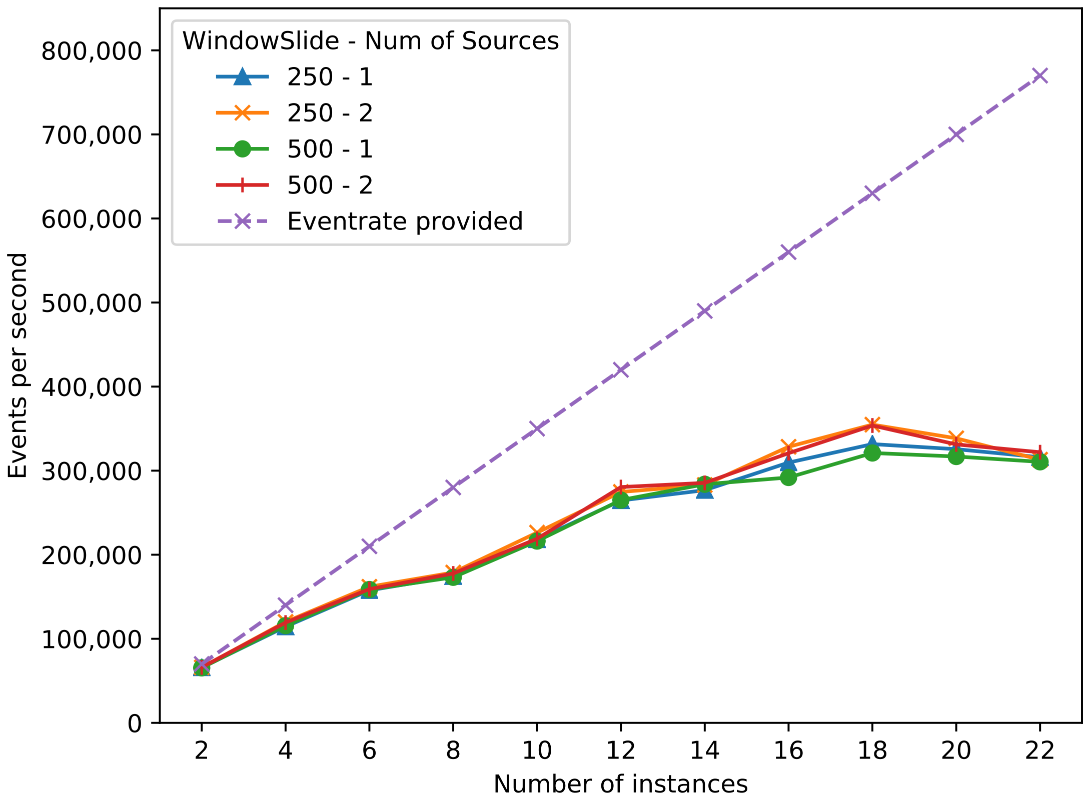


Figure 5.8: Throughput of Apache Flink with many slow sources.

Since Flink does not natively support ordering of multiple event streams, we did not order them. This decision would impact the framework performance only in a positive way. Since Flink cannot assign windows to instances based on a built-in scheduler, we decided to generate windows, then add equally distributed keys to the stream of windows and let flink assign the whole, keyed windows to instances. This turned out to be the most effective approach we found for windowing. We furthermore assured this approach did not impose a significant overhead compared to randomly assigning events to instances, we noticed a slowdown of 1.7% in the maximum speed.

PACE Figure 5.9 depicts the eventrates for the PACE framework. With a duplication factor of 2, the framework could keep up with the production rate until 4 sources, then the framework was overloaded. Note that the low rates after that point are due to the fact that the system was overloaded and they do not represent the actual maximum possible ratings with that configuration. For 2 instances the PACE framework scales linearly for all tested configurations.

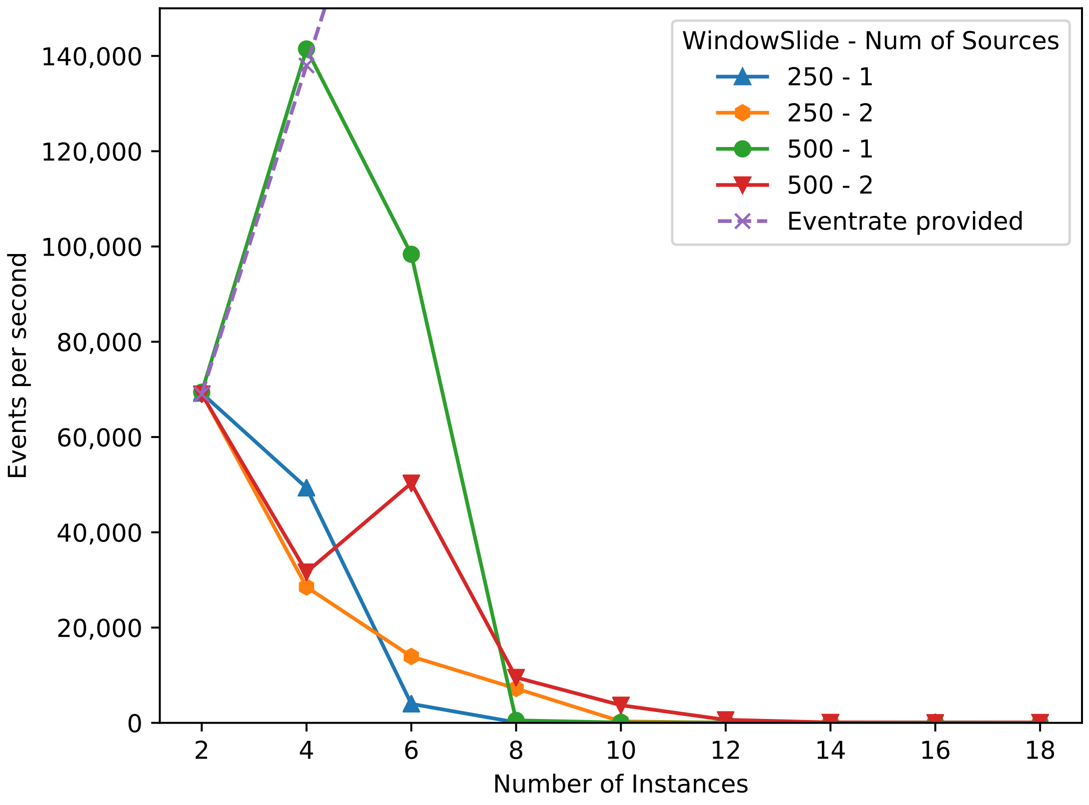


Figure 5.9: Throughput of PACE with many slow sources.

The maximum observed speed in this experiment was 141465 events per second, this is far below the maximum rate for a duplication factor of 2, in this case 201859.

This might be due to a design flaw in the PACE framework. The framework writes events to a buffer and immediately flushes this buffer after every event. The JVM then blocks until it has called the operating system function to send the data to the network-buffer, from where it gets sent via the wire. If the instance is busy doing its work, the network buffer might fill up because the instance won't read any data and therefore the JVM has to block until that data gets read. This would slow down the system and could be prevented by using another send buffer, that truly asynchronously sends the events.

SPACE Figure 5.10 shows the eventrates for the SPACE framework for this experiment. We added the maximum observed ratings for duplication found in experiment 5.4.3 for a factor of 2 and 4 as dotted lines. We observed proportional growth of the eventrate for the framework when running with a window slide of 500 and one source up to the maximum supported rate for this duplication factor, then it breaks down. When using a slide size of 250, the event rate is capped by the maximum supported rate of the respective duplication factor. The framework did not completely break down performance wise because of the implemented overload prevention mechanism. This mechanism did not work for 2 sources and a window slide of 250.

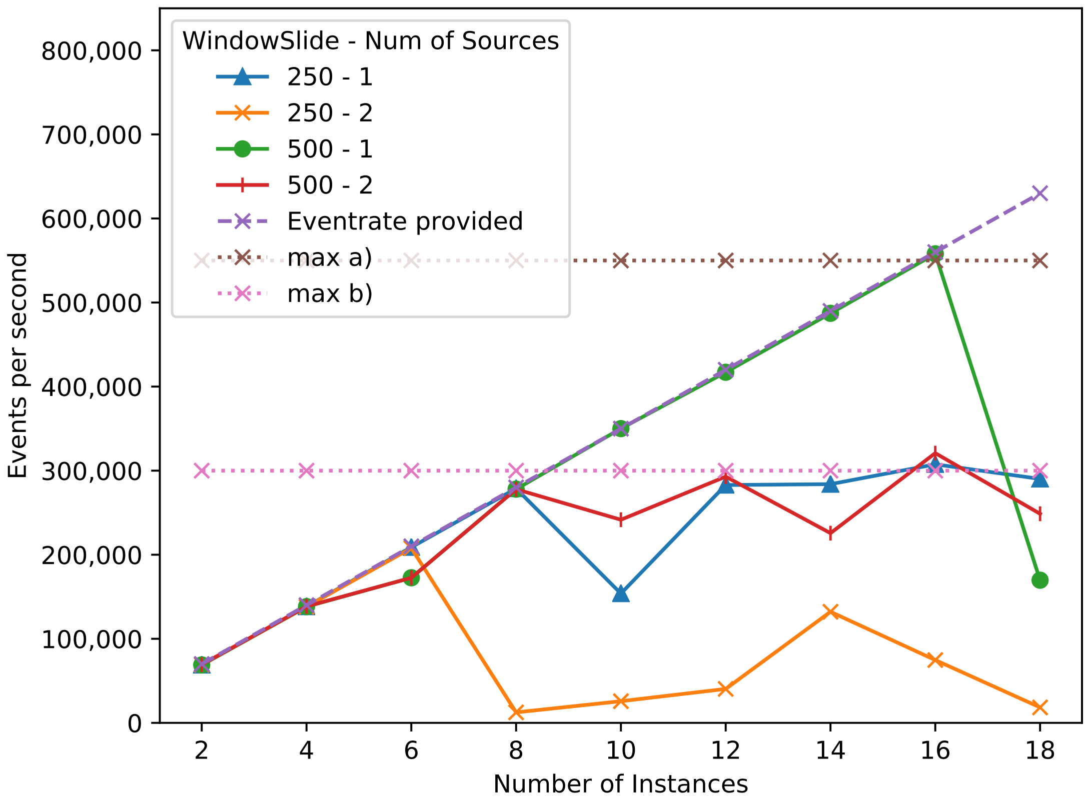


Figure 5.10: Throughput of SPACE with many slow sources.

5.5 Conclusion

The benchmarks have shown that the changes we made to the PACE framework significantly increased the performance of the resulting SPACE framework. Furthermore we increased the scalability and stability of the framework. The custom built SPACE framework outperformed Apache Flink in our benchmarks. The SPACE framework is able to scale nearly linear in a work scenario, up to the limits of its pipeline.

6 Related Work

Predictable Low-Latency Event Detection With Parallel Complex Event Processing Mayer et al. [MKR15] have shown that parallelization is necessary for scalable event detection. They studied CEP systems in the field of IoT scenarios and propose a partitioning scheme for event streams that changes depending on the workload, while maintaining low latency bounds. With the results of their work, they enable CEP systems to be used in IoT scenarios at scale.

RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing Balkesen et al. [BDWT13] have investigated run-based intra-query parallelism for CEP systems and have shown its benefits. This run-based approach resembles the data-parallel approach implemented in this work and hence represents the foundation upon which the PACE and hence the SPACE frameworks are built. Balkesen et al. [BDWT13] have shown the feasibility of their approach in multiple experiments and have proven great scalability.

Partition and Compose: Parallel Complex Event Processing In their paper, Hirzel [Hir12] have implemented state based parallel CEP and evaluated their results on real-world data-sets. They reached event rates of up to 830.000 events per second and observed nearly linear speedup for certain data sets (see [Hir12] Fig. 9) when parallelized. They have also shown that the achievable speedup by parallelization severely depends on the complexity of the query.

Integrating Scale Out and Fault Tolerance in Stream Processing using Operator State Management In their work, Fernandez et al. [FMKP13] have shown the feasibility of using stream processing in real-world cloud scenarios, when processing key-based data. Despite their focus is on optimizing resource consumption in pay-as-you-go scenarios, they are using key-based splitting of their data stream and observed nearly linear scalability of their processing system (see [FMKP13] Fig. 6). While key-based splitting provides great performance, it is not supported on all data sets and queries.

Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing Mayer et al. [MTR17] have investigated factors that influence latency in operator instances and propose methods for minimizing the latency in their system. In their paper, the authors describe the operator graph, the data parallelization, splitting and scheduling as implemented in the PACE framework.

Speculative Out-Of-Order Event Processing with Software Transaction Memory To speed up the processing of events on multi-core-machines, Brito et al. [BFSF08] have developed a speculative execution model to handle Out-Of-Order events. They have shown that even when the system has to discard some computations based on wrong speculations, there can be some speedup observed. Their system can, however, only be used on a single machine, without introducing large network load for state-synchronization.

Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams To increase the performance of stream processing systems, Balkesen and Tatbul [BT11] introduced several optimizations for Parallel Processing based on sliding windows. They, for example, propose to batch windows together, as later mentioned by [MTR17], to schedule these batches on nodes. Despite their performance gains, we did not incorporate these improvements into the SPACE framework.

Consistent Splitting of Event Streams in Parallel Complex Event Processing Siddam [Sid15] studied the PACE framework and implemented different consumption policies and investigated their behavior. There was no effort to drastically redesign the framework to increase its performance.

FlinkMan – Anomaly Detection in Manufacturing Equipment with Apache Flink Rivetti et al. [RBG17] have investigated Apache Flink for their use in anomaly detection and have shown some of its shortcomings. They implemented their anomaly detection based on Markov models and parallelized their processing by grouping the data based on keys incorporated in the data. This key-based parallelism is possible due to the independent processing of the data of individual machines under surveillance. Nonetheless, they needed to assure the data to be ordered, what is not supported by Flink. The authors hence had to implement their own reordering mechanism on top of the Flink primitives.

Furthermore, they noted that the network stack of Flink might be a bottleneck in their system because multiple operator instances running in one Flink worker share the same network stack.

7 Conclusion

7.1 Summary

The amount of data to be processed is growing, creating demand for more sophisticated and efficient data processing systems. In the area of low latency processing of events, Complex Event Processing is a promising approach. Since horizontal scaling is often more practical and cheaper than vertical scaling, parallelization and distribution are an important requirement for efficient and large scale processing with CEP.

In this thesis we described an existing framework for Complex Event Processing, PACE, in depth. We then analyzed its performance and have shown its shortcomings. We then proposed several improvements for the framework and implemented them. Thereby we created a new framework, SPACE, which was to evaluate. We compared the SPACE and PACE frameworks to each other and an established, industry-proven framework, Apache Flink.

While having flaws in certain conditions, the SPACE framework is capable of working with a higher rate of events than the other two frameworks. The improvements leading to the SPACE framework created a more than five-fold speedup of the maximum event rate, in certain scenarios even more. The SPACE framework has proven to scale up to at least 254 instances with non-overlapping windows and up to 16 instances if the windows do overlap and the instances are performing computationally intensive tasks.

Apache Flink was found to be the most sophisticated framework, providing lots of advanced features not used in this work. Whilst not being the fastest framework investigated, it is the most versatile one due to its more general approach.

7.2 Future work

Future work could include investigations on how to chain multiple OrderComponents together to achieve lesser performance penalties when ordering from multiple streams. Furthermore, event duplication with a factor greater than two could possibly be moved to multiple dedicated machines to circumvent the slowdown in the splitter.

To further increase the performance of the splitter in the SPACE framework, one could implement several proposed algorithms, like batching windows on instances.

This work assumed that the sources are producing events at nearly equal rates. If this assumption does not hold, one could investigate the feasability of sending periodic heartbeat messages from the sources to the splitter to indicate that it has not missed any events.

Bibliography

[BDR+16] L. Balogh, I. Dávid, I. Ráth, D. Varro, A. Vörös. “Distributed and Heterogeneous Event-based Monitoring in Smart Cyber-Physical Systems”. In: (Jan. 2016) (cit. on p. 11).

[BDWT13] C. Balkesen, N. Dindar, M. Wetter, N. Tatbul. “RIP: run-based intra-query parallelism for scalable complex event processing”. In: *Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS ’13*. ACM Press, 2013. DOI: 10.1145/2488222.2488257 (cit. on pp. 16, 17, 57).

[BFSF08] A. Brito, C. Fetzer, H. Sturzrehm, P. Felber. “Speculative out-of-order event processing with software transaction memory”. In: *Proceedings of the second international conference on Distributed event-based systems - DEBS ’08*. ACM Press, 2008. DOI: 10.1145/1385989.1386023 (cit. on p. 58).

[BT11] C. Balkesen, N. Tatbul. “Scalable Data Partitioning Techniques for Parallel Sliding Window Processing over Data Streams”. en. In: 8th International Workshop on Data Management for Sensor Networks; Conference Location: Seattle, WA, USA; Conference Date: August 29, 2011; . Association for Computing Machinery, 2011 (cit. on p. 58).

[CKS+15] P. Carbone, A. Katsifodimos, S. Sweden, S. Ewen, V. Markl, S. Haridi, K. Tzoumas. “Apache Flink™: Stream and Batch Processing in a Single Engine”. In: 38 (Jan. 2015) (cit. on p. 23).

[CM12] G. Cugola, A. Margara. “Processing Flows of Information: From Data Stream to Complex Event Processing”. In: *ACM Comput. Surv.* 44.3 (June 2012), 15:1–15:62. ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: http://doi.acm.org/10.1145/2187671.2187677 (cit. on pp. 13, 14).

[Cor14] I. D. Corporation. *The digital uni- verse of opportunities: Rich data and the increasing value of the Internet of Things*. 2014. URL: http://www.emc.com/leadership/digital-%20universe/2014iview/internet-of-things.htm (cit. on p. 11).

[FMKP13] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, P. Pietzuch. “Integrating scale out and fault tolerance in stream processing using operator state management”. In: *Proceedings of the 2013 international conference on Management of data - SIGMOD ’13*. ACM Press, 2013. DOI: 10.1145/2463676.2465282 (cit. on p. 57).

[Fou18a] T. A. Foundation. *The Pattern API*. 2018. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/libs/cep.html#the-pattern-api (cit. on p. 23).

[Fou18b] T. A. S. Foundation. *Flink Architecture.* 2018. URL: https://flink.apache.org/flink-architecture.html (cit. on pp. 23, 24).

[Fou18c] T. A. S. Foundation. *Flink Data Sinks.* 2018. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.6/dev/datastream_api.html#data-sinks (cit. on p. 15).

[Fou18d] T. A. S. Foundation. *Powered by Flink.* 2018. URL: https://cwiki.apache.org/confluence/display/FLINK/Powered+by+Flink (cit. on p. 23).

[Fou18e] T. A. S. Foundation. *The ProcessFunction.* 2018. URL: https://ci.apache.org/projects/flink/flink-docs-release-1.5/dev/stream/operators/process_function.html (cit. on pp. 23, 24).

[Fri16] T. K. Friedman Ellen. *Introduction to Apache Flink.* 2016 (cit. on p. 23).

[Gög18] J. Göggel. "Vergleich und Analyse geläufiger CEP Systeme". Universität Stuttgart, Feb. 3, 2018 (cit. on p. 23).

[Hed17] U. Hedtstück. *Complex Event Processing.* Springer Berlin Heidelberg, 2017. DOI: 10.1007/978-3-662-53451-9 (cit. on pp. 14, 19).

[Hir12] M. Hirzel. "Partition and compose". In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems - DEBS '12.* ACM Press, 2012. DOI: 10.1145/2335484.2335506 (cit. on p. 57).

[KMR+13] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, M. Völz. "Rollback-recovery without checkpoints in distributed event processing systems". In: *Proceedings of the 7th ACM international conference on Distributed event-based systems - DEBS '13.* ACM Press, 2013. DOI: 10.1145/2488222.2488259 (cit. on pp. 14, 15).

[MKR15] R. Mayer, B. Koldehofe, K. Rothermel. "Predictable Low-Latency Event Detection With Parallel Complex Event Processing". In: *IEEE Internet of Things Journal* 2.4 (Aug. 2015), pp. 274–286. DOI: 10.1109/jiot.2015.2397316 (cit. on pp. 14, 16, 57).

[MSPC12] D. Miorandi, S. Sicari, F. D. Pellegrini, I. Chlamtac. "Internet of things: Vision, applications and research challenges". In: *Ad Hoc Networks* 10.7 (Sept. 2012), pp. 1497–1516. DOI: 10.1016/j.adhoc.2012.02.016 (cit. on p. 11).

[MST+17] R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, U. Ramachandran. "SPECTRE". In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference on - Middleware '17.* ACM Press, 2017. DOI: 10.1145/3135974.3135983 (cit. on p. 23).

[MTR17] R. Mayer, M. A. Tariq, K. Rothermel. "Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing". In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17.* ACM Press, 2017. DOI: 10.1145/3093742.3093914 (cit. on pp. 14–17, 19, 57, 58).

[NM02] N. Nagarajayya, S. Mayer. *Improving Java Application Performance and Scalability by Reducing Garbage Collection Times and Sizing Memory Using JDK 1.4.1.* Nov. 1, 2002. URL: https://www.oracle.com/technetwork/systems/index-156457.html (cit. on p. 32).

[NRNK10] L. Neumeyer, B. Robbins, A. Nair, A. Kesari. “S4: Distributed Stream Computing Platform”. In: *2010 IEEE International Conference on Data Mining Workshops*. IEEE, Dec. 2010. DOI: 10.1109/icdmw.2010.172 (cit. on p. 14).

[PBYP17] B. Petersen, H. Bindner, S. You, B. Poulsen. “Smart grid serialization comparison: Comparision of serialization for distributed control in the context of the Internet of Things”. In: *2017 Computing Conference*. IEEE, July 2017. DOI: 10.1109/sai.2017.8252264 (cit. on pp. 32, 35).

[RBG17] N. Rivetti, Y. Busnel, A. Gal. “FlinkMan”. In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems - DEBS '17*. ACM Press, 2017. DOI: 10.1145/3093742.3095099 (cit. on p. 58).

[SGN+11] S. Suhothayan, K. Gajasinghe, I. L. Narangoda, S. Chaturanga, S. Perera, V. Nanayakkara. “Siddhi”. In: *Proceedings of the 2011 ACM workshop on Gateway computing environments - GCE '11*. ACM Press, 2011. DOI: 10.1145/2110486.2110493 (cit. on p. 16).

[Sid15] N. Siddam. “Consistent Splitting of Event Streams in Parallel Complex Event Processing”. English. Master Thesis. University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, Germany, Oct. 2015, p. 73. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=MSTR-2020004&engl=1 (cit. on p. 58).

[SS13] O. Saleh, K. U. Sattler. “Distributed Complex Event Processing in Sensor Networks”. In: *2013 IEEE 14th International Conference on Mobile Data Management*. Vol. 2. June 2013, pp. 23–26. DOI: 10.1109/MDM.2013.60 (cit. on p. 11).

[Tay10] R. C. Taylor. “An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics”. In: *BMC Bioinformatics* 11.Suppl 12 (2010), S1. DOI: 10.1186/1471-2105-11-s12-s1 (cit. on p. 13).

[VK18] J. Vanura, P. Kriz. “Perfomance Evaluation of Java, JavaScript and PHP Serialization Libraries for XML, JSON and Binary Formats”. In: *Services Computing – SCC 2018*. Springer International Publishing, 2018, pp. 166–175. DOI: 10.1007/978-3-319-94376-3_11 (cit. on p. 32).

[WLLB06] F. Wang, S. Liu, P. Liu, Y. Bai. “Bridging Physical and Virtual Worlds: Complex Event Processing for RFID Data Streams”. In: *Advances in Database Technology - EDBT 2006*. Ed. by Y. Ioannidis, M. H. Scholl, J. W. Schmidt, F. Matthes, M. Hatzopoulos, K. Boehm, A. Kemper, T. Grust, C. Boehm. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 588–607. ISBN: 978-3-540-32961-9 (cit. on p. 11).

[ZK15] E. Zamecnikova, J. Kreslikova. “Comparison of platforms for high frequency data processing”. In: *2015 IEEE 13th International Scientific Conference on Informatics*. IEEE, Nov. 2015. DOI: 10.1109/informatics.2015.7377850 (cit. on p. 23).

All links were last followed on September 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature