Data Stream Operations as First-Class Entities in Component-Based Performance Models*

Dominik Werle[0000−0002−2430−2578], Stephan Seifermann, and Anne Koziolek[0000−0002−1593−3394]

Karlsruhe Institute of Technology (KIT), 76131 Karlsruhe, Germany
{dominik.werle,stephan.seifermann,koziolek}@kit.edu

Abstract. Data streaming applications are an important class of data-intensive systems. Performance is an essential quality of such systems. It is, for example, expressed by the delay of analysis results or the utilization of system resources. Architecture-level decisions such as the configuration of sources, sinks and operations, their deployment or the choice of technology impact the performance. Current component-based performance prediction approaches cannot accurately predict the performance of those systems, because they do not support the metrics that are specific to data streaming applications and only approximate the behavior of data stream operations instead of expressing it explicitly. In particular, operations that group multiple data events and thus introduce timing dependencies between different calls to the system are not represented sufficiently. In this paper, we present an approach for modeling networks of data stream operations including their parameters with the goal of predicting the performance of the resulting composed data streaming application. The approach is based on a component-based performance model with queueing semantics for processing resources. Our evaluation shows that our model can more accurately express the behavior of the system, resulting in a more expressive performance model compared to a well-encapsulated component-based model without data stream operations.

Keywords: data streaming, performance modeling, component-based software engineering, complex event processing

1 Introduction

Systems that process large amounts of data from varied sources have become an important class of software systems in recent years. Reasons for this development are the vastly increasing amount of data sources from which data is gathered and improvements in methods for data analysis.

From a software engineering viewpoint, building such software systems entails specific challenges in the different activities that are part of the software

* Partially funded by the German Research Foundation (DFG) as part of the Research Training Group GRK 2153: "Energy Status Data – Informatics Methods for its Collection, Analysis and Exploitation"

engineering process [8]: for example planning which types of processing hardware are required for the system and how the system will behave in future scenarios where the number of data sources or other data characteristics change.

In this paper, we address the problem that current component-based performance models to our knowledge cannot express stateful operations which are commonly used in data streaming applications. For example, when data is collected and emitted as a group when a data item with specific characteristics arrives or when a specified duration has passed introduce timing dependencies between calls. Expressing such state requires modeling workarounds that break the encapsulation and thus hinder the separate reusability and maintainability of components [21].

There are different paradigms for building systems that process large amounts of data. In the context of this paper, we will focus on applications that process continuously arriving streams of data (*streaming applications*), as opposed to applications that regularly process larger batches of data.

Overall, we present a new approach for modeling the performance of data streaming applications. More specifically, we provide a method for expressing the interaction between components that process data streams in performance models and a simulation approach for analysing these models. This article extends a previously presented sketch of the approach [21] with a more thorough discussion of the modeling approach and its relation to component-based performance models, as well as an implementation and initial evaluation of the approach.

The main stakeholders of the approach are the software engineers that have to make decisions about the configuration of the system, for example, suitable sizes for sliding windows. Furthermore, we target the operations engineers that have to predict how the system will scale with expected changes in the load or additional analyses on the data. We consider both groups by incorporating the decisions they have to make in the modeling language, thus allowing them to make what-if analyses using the presented approach.

Subsequently, the main goal of our approach is to support software engineers that build data streaming applications with suitable modeling and quality prediction tools. To this end, this article addresses the following two research questions.

RQ_1: What is a suitable way of representing data stream operations in component-based performance models while keeping components and data stream operations reusable and parameterizable?

RQ_2: How can the behavior of data stream operations be incorporated into simulations of otherwise stateless component-based performance models?

RQ_1 is targeted at the modeling language itself. It is focused on identifying the relevant operations and how they can be included in a modeling language in a composable way that can be used in combination with current component-based performance modeling tools. The second question, RQ_2, focuses on the way the models are then analysed. Particularly, the simulation of the system behavior that includes data stream operations needs to be able to interface with current simulation approaches. We present three contributions towards these questions:

C$_1$: An approach for representing operations in a composable, architecture level performance model,

C$_2$: A simulation approach for data stream operations,

C$_3$: An evaluation of the simulation for a case study system.

2 Running Example

To illustrate our approach, we use a running example that is an adaptation of the 2014 grand challenge of the conference on Distributed and Event-Based Systems (DEBS 2014) [10]. The example application processes meter readings that smart plugs send to a system to calculate an outlier score for houses.

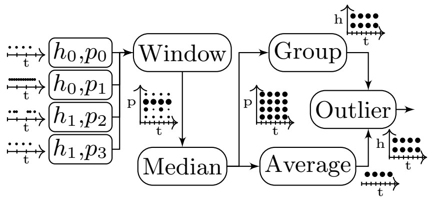

Fig. 1: Illustration of the running example (Source: [21]).

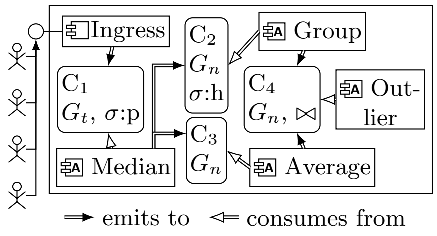

Fig. 2: Simplified illustration of the performance model (Source: [21]).

The example system is illustrated in Figure 1. The following more detailed explanation is partially taken from [21, sec. 3], where we initially introduced the example. Figure 2 shows an implementation of the system in our modeling approach and will be discussed in detail Subsection 5.4. N smart plugs send data to the system. Each of the smart plugs belongs to one of H households. A plug with id j that belongs to household i is named h_i, p_j in the illustration. The rate at which plugs send data can differ for different plugs and can vary over time. *Window* creates data windows of time length S. The data is collected grouped by the plug id, i.e., *Window* emits a data window for each plug and each point of time that windows are created. The windows are created every Δ time units, at the points of time $T_i = \Delta \cdot i, i \in \mathbb{N}$, resulting in N newly created windows for each point in time T_i. Windows overlap if $\Delta < S$. Then, sensor readings are included in multiple windows. As a result, every window spans the S time units prior to its creation. *Median* creates a median for each window and plug, resulting in N (number of smart plugs) medians for each T_i. *Average* collects all medians for one T_i and calculates one overall average value of all medians for each T_i. *Group* collects all median values for one household for one T_i by collecting all median values of all plugs for the T_i and regrouping them by the household id. For each of the H households, *Outlier* calculates the ratio of readings of plugs inside the household that are greater than the overall average and emits this value as the outlier value of the household, resulting in H values for each T_i. The metric of interest for this system is the time between the creation of a date in the plug and its first appearance in the result of an outlier calculation, the *delay*.

Interesting questions that the architect of the system may want to answer are how well the system scales if the number of plugs increases, and which resources are needed for a particular load. Other questions can be more closely related to the actual functionality of the system: How does changing the shift and size of the windowing operation change the delay? How does the association between plugs and households change the quality of the system (i.e., small amount of households with many plugs each in comparison to large amount of households with little plugs each)?

3 Background

In this section, we present relevant foundations for our approach regarding the way component-based performance models are constructed and used.

3.1 Component-based Performance Models

Component-based performance models allow a decomposition of systems into so-called components. An example for a modeling language that allows this is the Palladio Component Model (PCM) [17], which we have based our approach on. The model defines interfaces that describe a collection of services. A component can either require or provide interfaces and describes the observable effect regarding performance for each provided service. Particularly, components can call other components' services. A system model is composed of components that together provide services to the user of the system. Based on its modeling primitives, there are extensions to the model and its analysis that add capabilities for simulating additional behavior of the system that are relevant for its performance such as virtualization, network protocols or different types of hardware resources and operating system schedulers. A usual way of evaluating models is to map resources to queues and serving nodes that process items in those queues. This network of queues is then either statically analysed or simulated.

Performance models are used to evaluate different performance metrics, such as the response time of systems to requests of different types and the utilization of resources. It should be possible to model, calibrate and reuse the different parts of the model independent from each other. This allows simple reuse and extension of models. Additionally, the system should be described in a way that exposes all relevant design decisions to the architect.

Parametric dependencies additionally allow effect descriptions to be parametrized regarding characteristics of the environment of the call. An example for this environment are parameters that are passed into a method. The concrete values that these parameters take depend on the composition of the system from components and are usually resolved when the system is analysed or simulated. This approach enables both the reuse and exchange of components, as well as a change of the workload if parameters are also used at the system interface.

3.2 State in Performance Models

A basic idea for component-based architecture models is that they are *stateless*. This means that the *description* of the service effects cannot refer to the current state of the system, its resources, or other components, but the *analysis* will implicitly keep state and schedule or parametrize service effects accordingly. Therefore, the interaction between different calls to the system is clearly encapsulated in so-called resources. The PCM distinguishes two types of resources. *Active* resources represent for example CPUs or HDDs. They can be represented as queues and service places in queueing models. Incurring a resource demand in a process blocks and thus delays the process until the resource has satisfied the demand. If multiple processes request the same resource at the same time, there needs to be some form of queueing or scheduling. *Passive* resources behave similar to semaphores. They have a specified token count and tokens can either be acquired or released. If a process wants to acquire a token when there is none left, it is blocked until another process releases a token. To summarize, while the systems that are modeled have state at runtime, the model of the components themselves cannot refer to this state in an explicit way, for example by branching depending on the current load of the system. The only way to depend on the current state of the system is through resources.

3.3 Stochastic Expressions and Dependencies

Performance models use stochastic expressions to describe stochastic processes that occur in the system or in its usage. A common example for this is the behavior of users of the system. Furthermore, stochastic descriptions can be used for aspects of the system that are not modeled in detail, for example when garbage collection happens in the Java Virtual Machine. If stochastic expressions are used with parametric dependencies, they can be used to express *stochastic dependencies* regarding the performance of the system.

3.4 Challenges for Modeling Data Streaming Applications

In this section, we discuss the major challenge for modeling data streaming applications using component-based performance models: representing timing dependencies between calls to the system.

Data streaming applications usually describe the behavior of the system depending on incoming calls that deliver data at the system interface. In the following, we use the term *data events* for those calls. In our work, we consider data *stream* operations that are aligned with the well-established CQL continuous query language [2]. In contrast to business information systems that provide services that users can call, data stream operations are not described in terms of single requests that are handled independently. Instead, they operate on multiple requests. For example, data events may be grouped in windows and then processed further as a group instead of single elements, which introduces a delay for single data events that depends on other events or operation characteristics.

Secondly, stochastic descriptions of resource demands cannot be stochastically dependent at different points of the system for one data event, if system models do not operate on groups of data. For example, if a particularly hard to calculate data point arrives at the system, high resource demands might incur at different points of the system in succession, resulting in an outlier of the calculation time for this data event. However, if these stochastic dependencies are not made explicit across regrouping or joining of data, resource demand descriptions in the model are stochastically independently from each other and thus may lead to loss of accuracy of the model. The need for suitable state abstractions in component-based performance models has been previously identified and discussed in the context of messaging systems by Happe et al. [7].

In current modeling approaches, if architects desire to model a data streaming application, they have to use a model of the system that does not reflect the actual structure and behavior of the system, but an approximation that behaves similarly regarding the performance metric of interest. For example, data events that are grouped into sliding windows inside the system might have to be modeled by modeling the arrival of windows instead of single data events at the system boundary. As a consequence, the performance-related characteristics of windows, such as the number of contained elements, have to be modeled manually instead of being derived automatically by the model.

4 Approach

In this section we introduce the modeling concepts that are provided in our approach in detail. This section addresses RQ_1 (see Section 1): *What is a suitable way of representing data stream operations in component-based performance models while keeping components and data stream operations reusable and parameterizable?*

4.1 Modeling Concepts

In this section, we introduce the modeling concepts that are novel in our approach and their role in modeling data streaming applications. An overview of the concepts is illustrated in Figure 3. The illustration is explained in detail in the following subsections.

We currently do not support the full set of operations for a streaming application (as for example in CQL [2] or in LINQ [15]). While our implementation focuses on operations that are required for our case study system, our simulation is implemented to be extensible for additional types of operations that change the stream of data that flows through the system.

Data Events. A *data event* describes the context of a call that is passed through the system in our approach. In our model, this context has an identity and can be passed between different calls.

Each data event is explicitly created by a process in the system. In addition to an implicit *date of birth*, the architect can also specify other characteristics that are relevant for the performance of the system or for the distribution inside data stream operations, for example because the characteristic influences how elements are grouped. Each characteristic is described via an expression and it is fixed when the data event is created. The expression can be stochastic, i.e., be described as a distribution function. Then, a value from this distribution is generated when the value is fixed. Additionally, the expression can depend on other parts of the current call context, such as parameters of the current call.

An example for a characteristic is the plug id of sensors that a data event is created from and then sent to the system. It is either specified as a distribution function in the workload that is applied to the system or is the result of different workload drivers (sensors) with fixed sensor types that send data to the system simultaneously. At some point in the system, the data events are then grouped by the type of sensor, resulting in different sizes of groups that have to be further communicated and processed in the system.

Note that a data event is separate from the notion of call flow through the system. While a workload might specify calls to system interfaces that pass data events to the system, this data event might then be picked up and further processed by other calls or recurring processes in the system. Thus, the data event carries an identity and fixed characteristics through the system. This identity and the attached date of birth of the data event can then be used for delay measurements in the system by specifying a point in the system where the current age of a data event is to be recorded. This information is then collected and presented to the performance analyst as a result of the analysis. Subsequently, our definition of data events addresses the challenge of parametric dependencies across calls (see Subsection 3.4).

Data Channels. The main novelty in our approach are so called *data channels*. The concept is based on event channels as introduced by Rathfelder [16] which are, however, mapped to normal call semantics for a system. We extend the idea of event channels by additional state and active processing inside the channel. A data channel is a type of resource that encapsulates state regarding data events. When modeling a system, data can be either *emitted to* or *consumed from* a data channel.

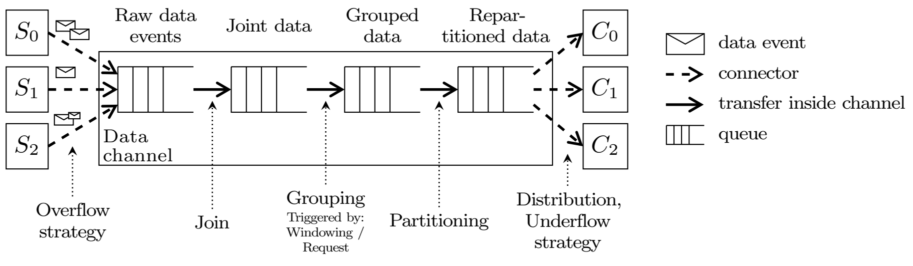

Fig. 3: Illustration of the model of a data channel. Data events that are emitted to the data channel by components pass through different queues.

Overall, a data channel can be seen as a series of queues that data events emitted to the channel are passed through, as illustrated in Figure 3. The steps between queues have an associated *operation* and a *trigger*. When triggered, the step takes one or multiple elements from the previous queue and emits a new element to the next queue. For the first queue (*raw data events*, Figure 3), elements are directly added by emitting components. For the last queue (*Repartitioned data*), consuming components can take data events. Our current implementation supports the following steps: *join*, *grouping* and *partitioning*. In principle, this can be extended to further steps, if required for different types of operations.

In the following we discuss how a data channel can be configured regarding the following aspects, in order of processing: 1) sinks, sources and connectors, 2) capacity, 3) over-/underflow discipline, 4) joining, 5) grouping, 6) partitioning, distribution strategy. Together, the configuration and the semantics contribute towards allowing timing dependencies in the model (see Subsection 3.4).

Sinks, Sources and Connectors. A component can communicate with the data channel via a connector. A connector can be either a sink or a source connector, depending on whether it connects the channel to a source (i.e., a component that emits data to the channel), or a sink (i.e., a component that consumes data from the channel). Sink connectors can either be *pushing* or *polling*. If the connector is pushing then a process is immediately triggered when new data is available and starts a service that is associated with the given connector inside the component. The parameter to these service calls is the data event. If the connector is polling then data events can be actively consumed from any service inside the component.

Capacity. A data channel can define a capacity of data events it can keep at a point of time. If no capacity is defined, there is no limit to this number. The capacity is defined in terms of the number of data events, not their size; this would however be a possible extension to our concept, if required. Currently, our notion of capacity only considers total raw data events that are put into the data channel and have not been processed further yet; however, this can be extended to the total number of data elements currently residing in the channel, or other notions of capacity. If an incoming data event would surpass the capacity of the data channel, the over-/underflow discipline determines how the call proceeds.

Over-/Underflow Discipline. The channel specifies an *overflow discipline* that determines whether in case of an incoming data event that would surpass the capacity of the data channel, 1) an incoming data event is dropped, 2) it replaces the element that was last added to the channel, 3) the first element that would be emitted is dropped and the new element is added, or 4) the call that wants to emit an data event to the channel is blocked until the capacity is not reached anymore. In the latter case, the waiting calls are handled in the order they try to emit data to the channel, i.e., the first call that wants to emit data to the channel can do this first after data has been removed from the channel and there is free capacity again.

For consuming calls, the channel specifies an *underflow discipline* that determines how calls that want to consume data from the channel when there is none available should be handled. The available options are to either 1) return without data elements, or 2) block until data is available. In the latter case, similar to calls waiting to emit data to the channel, the calls are handled in the order they try to consume data from the channel.

The described disciplines focus on the capacity of the channel and do not address other reasons for not accepting data elements, such as elements that arrive too late at a channel, which we are planning to address in future work.

Joining. Joining regroups elements on a given join condition. In our implementation, joins are realized by tagging data with information about the incoming connectors. A joint data element is emitted to the *Joint data* queue, when a data event for each connector is available. The modeler specifies for each incoming connector whether a data event can contribute multiple times to joint data events or only once. If it can contribute multiple times, the last data event for each incoming connector is not removed but can be used multiple times; previous elements are removed if a new element arrives. If data events for a connector can only contribute once, they are removed from the data channel upon inclusion in a joint date.

Grouping. In general, a grouping operation retains elements until a trigger occurs. This trigger is either 1) based on the time in the system, 2) on characteristics of the data in the group, or 3) or on an external trigger. Our implementation currently provides three types of grouping, which are examples for the respective types of triggers: 1) sliding windows, 2) holdback grouping, and 3) consume-all grouping. We explain these types in the following in more detail to illustrate the principle of grouping channels.

Sliding windows are a concept where elements are grouped in windows at periodic points of time. Usually, a sliding window operation is configured with a *size* S and a *shift* Δ. Every Δ time units, the windowing operation emits all data events that have arrived in the last S time units as a grouped data event. This means that windows overlap if $\Delta < S$. Then, data events are included in multiple windows.

Holdback grouping is a type of grouping, where elements are grouped according to a key function. Elements with the same key are collected in a group and held back. The channel can keep a given number of groups at the same time (default is one). If an additional key is discovered in an arriving data event for whose calculated key function currently no group that is held back exists, a new one is created. If the number of groups then exceeds the specified maximum, the oldest group that is held back is emitted. Another example for grouping based on characteristics of the data in the group would be to emit a group of elements when a specified number of elements are reached.

Consume-all grouping describes the idea that elements are collected and emitted as a whole as soon as a trigger happens, such as a consume action.

Partitioning. *Partitioning* is an example for an operation that further operates on data that is already grouped. Particularly, it is regrouped according to a key function. This means that for every element in the input group of data events, the key function is calculated. Then, for each distinct value of the key function, a grouped data event is created that contains all elements that have mapped to this key. In our running example, this occurs when medians have been calculated for all plugs for a window and then are regrouped according to the household the plug belongs to. Note that for this to work, first all elements in the window, i.e., all calculated medians for all plugs in the sliding window, have to be collected and then regrouped according to the key function household id.

Distribution Strategy. Distribution strategy describes whether elements 1) are distributed to all target connectors, 2) are passed out to the connectors in succession (round-robin), or 3) the first connector to request data (for polling consumers) gets the next data event from the data channel.

5 Evaluation

In this section, we show how the presented approach can support software architects in predicting the performance of the system by explicitly representing its expected behavior in a performance model.

5.1 Evaluation Question

The evaluation presented in this section addresses RQ_2 (see Section 1): *How can the behavior of data stream operations be incorporated into simulations of otherwise stateless component-based performance models?* The derived evaluation question is: *Is it feasable to model and simulate the timing behavior of stateful data-stream operations using our approach?*

To answer this question we discuss how our modeling approach applies to the system introduced in Section 2. We first describe the workload and configuration of our system in our experiment (Subsection 5.2). In Subsection 5.3 we discuss the metric of interest and the expected behavior of the system regarding this metric. We subsequently show, how we can build a model that predicts its performance (Subsection 5.4). We then discuss the results and benefits of the simulation. Our evaluation provides initial evidence that we can use the presented approach to model relevant data streaming applications. The evaluation of the accuracy of our modeling approach regarding measurements from a real system and the comparison with other modeling approaches is subject to future work and not addressed in this evaluation.

5.2 Experiment Setup: Workload and Configuration

We have created a workload that is based on the first ten minutes of the data from the house with house id 0 from the DEBS 2014 grand challenge data[1]. As a result, our workload represents 14 households with 107 plugs total. While our approach can handle larger experiments, we chose a small excerpt of the data to keep the following presentation easy to understand. To provide further evidence

[1] The data is available publicly via the website of the challenge [9].

about the limits of the scalability of our approach, both for the simulation as well as for building the models, we need to model larger systems in future work and for more complicated workloads. On average, around 45.775 data points arrive at the system interface per second. For our chosen window size ($S = 25\,\mathrm{s}$), we can expect an average number of total data points per window of $R = 25 \cdot 45.775 = 1144.375$. The shift of windows is chosen as $\Delta = 50\,\mathrm{s}$.

5.3 Experiment Setup: Metric and Performance

We assume that in the case study system, the system architect is interested in the *delay* of data events when they appear in the ouput of the outlier detection for the first time. This delay of the system is formed as follows. Each data event has to be sent from the sensor to the system, resulting in a network delay. For each data event, windowing with size S and Δ leads to a delay that is uniformly distributed in $[0, S]$. The number of elements in the window is determined by the number of data events that arrive during this window, which depends on the usage scenario. Sensor readings that arrive just before a window is emitted are only negligibly delayed by the windowing, data points that arrive just after a window has been created are delayed by $25\,\mathrm{s}$. Calculating the median takes a resource demand that depends linearly on the number of elements inside each window, or log-linear resource demand if implemented naively using sorting. This number of elements depends on the arrival patterns of data events. In our example, we chose every resource demand as $100 + 100 \cdot n$, where n is the number of elements in the grouped data element. Regrouping the elements results in a resource demand that depends on the number of elements to regroup. In this example this is the number of plugs. The resource demand for calculating the average of all medians inside a time window depends linearly on the number of medians, i.e., the number of plugs. This number depends on the usage scenario. The calculation can only be triggered if the system determines that all relevant medians have arrived at the component. To do this, the system detects when a median for the next window has arrived. This leads to an additional delay of $\Delta = 50\,\mathrm{s}$. The outlier detection has to wait for both the overall average and the household-wise regrouped elements. For each joint element, the calculation resource demand then depends on the number of plugs in the given household.

Additionally, communication delays between components occur and become relevant if the operations are distributed across resources. In our example, every operation is implemented as a single component. For each component there runs one process that takes all available input data, processes it and then waits for additional input data.

If we only consider the timing dependencies introduced by collective operations in the system, we can expect a delay inside $[50\,\mathrm{s}, 75\,\mathrm{s}]$: The sliding window contributes $0\,\mathrm{s}$ to $25\,\mathrm{s}$ to the delay, waiting for one window shift due to grouping contributes an additional delay of $50\,\mathrm{s}$.

For a processing rate of $P = 10\,000$ resource demand units per second and no contention at active resources (i.e., resources can serve infinitely many requests in parallel), we assume about an additional delay of $2 \cdot (100 + 100 \cdot R)/P = 22.91\,\mathrm{s}$ (R

as defined in Subsection 5.2): The creation of medians and the outlier calculation each have to pass over all data points in the window. As a result, we expect an overall delay in $[72.91\,\mathrm{s}, 97.91\,\mathrm{s}]$.

5.4 Experiment Setup: Model

In this section, we present how our approach can be applied to the example system. We have implemented the parts of the approach that are required to model this system and make on-going implementation publicly available [19]. All artefacts used in our evaluation and a guide on how to extract the relevant data from the DEBS data set and run the simulation are available online [20].

Figure 2 illustrates a realization of our running example in PCM. The following is an extended explanation based on previous work where we have introduced a sketch of this model [21, sec. 4]. The component *Ingress* handles the sensor reading ingress and writes data to data channel C_1. There is a usage scenario for each sensor which calls *Ingress* with a characterization of its plug and household id. *Ingress* then creates a date with the specified characteristics (thus also implicitly creating a birth date) and emits it. The windowing of readings is specified in the data channel C_1. In our example setup, the data channel creates windows of size 25 s every 50 s. This means that windows are created at points of time $T_i = 50\,\mathrm{s}, 100\,\mathrm{s}, 150\,\mathrm{s}, \mathrm{etc.}$ and span all data points that arrive during $[25\,\mathrm{s}, 50\,\mathrm{s}), [75\,\mathrm{s}, 100\,\mathrm{s}), [125\,\mathrm{s}, 150\,\mathrm{s}), \mathrm{etc.}$ respectively. For every other component, there exists an additional usage scenario (and interface to trigger the component's processing) that repeatedly triggers the component after it has finished its current processing. Components for which such a scenario exists are depcited with the symbol ▤. This means that for each of the components *Median*, *Group*, *Average* and *Outlier*, there is a process that tries to consume data from the respective data channel. If necessary, the process waits until data is available. Then it processes the data. The processes take an amount of time that depends on the number of elements that are processed. Each of the processes then emits data or, in case of *Outlier*, ends the processing and measures the *delay* of all data points in the currently processed group of elements.

For each plug and each T_i, *Median* consumes a window from C_1, possibly blocking until a window is available. It then emits to C_2 and C_3. C_2 groups depending on window start and end and partitions according to the household id. C_2 emits the group when it discovers that the window has changed, thus delaying the processing chain by one window size. *Group* consumes from C_2 and emits to C_4. C_3 again groups by the window start and end. *Average* consumes groups of medians for each time window from C_3 and emits to C_4. C_4 joins data from *Group* and *Average* based on the start/end of windows. Since the overall average of a window is only calculated once, it can contribute arbitrarily often to the join. As a result, C_4 contains a joint date for each *Outlier* consumes from C_4 and specifies an appropriate resource demand.

We have generated a usage model from the DEBS data set, which results in an additional usage scenario for each of the 107 plugs. Each of the users calls the system interface that is delegated to *Ingress* and provides its household id and plug id as parameters to the call. We create a distribution of times between readings for each plug and use this distribution as the time between readings in our model.

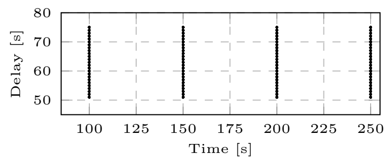

Fig. 4: Delay for data events that arrive in the analysis component in our evaluation system.

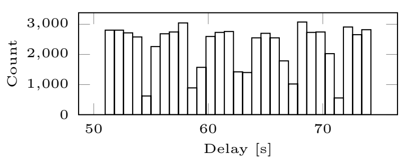

Fig. 5: Histogram of delays that are measured in the simulation for negligible processing times (cf. Figure 4).

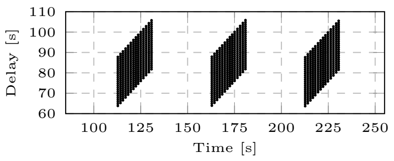

Fig. 6: Delay for data events that arrive in the analysis component in our evaluation system.

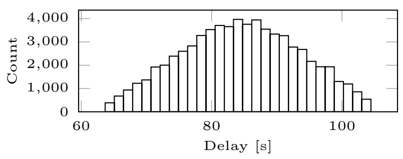

Fig. 7: Histogram of delays that are measured in the simulation for non-negligible processing times.

5.5 Results

Our simulation reproduces the expected number of arriving elements described in Subsection 5.2 ($R = 1144.375$). Using the simulation we can derive the previously described metrics about the system. We first focus on the case where delay introduced by processing can be neglected and only consider temporal dependencies. Therefore, all data points and groups are processed as soon as they are available. The delay at the point of measuring (after the outlier calculation is finished) is depicted in Figure 4. The distribution of delays across the simulation is displayed in Figure 5. The results of the simulation with non-negligible processing times is depicted in Figure 6 and Figure 7. This shows a scenario, where processing of data groups uses resources and thus leads to additional delay.

5.6 Discussion

As can be seen from the results, our model can represent the expected number of arriving elements in the sliding window. Furthermore it predicts delays in $[51\,\mathrm{s}, 75\,\mathrm{s}]$ with an average of 63 s and thus accurately reproduces the expected delay for negligible processing times in $[50\,\mathrm{s}, 75\,\mathrm{s}]$ (midpoint at 62.5 s). If processing times are not negligible, the resulting additional delay leads to a change of the distribution of the delay. The results of the simulation are in the interval $[63.67\,\mathrm{s}, 105.85\,\mathrm{s}]$ with an average of 84.66 s as compared to the manually deducted estimate in $[72.91\,\mathrm{s}, 97.91\,\mathrm{s}]$ (midpoint at 85.41 s). Creating the estimate as discussed in Subsection 5.3 requires manual effort for deriving how which part of the system influences the behavior and interacts with the type of data it receives instead of plugging the system together from components and simulating the actual behavior. This is particularly challenging, if the influence of different parts changes due to changes in the configuration of the system. For example, if the constant factors in the resource demands are relevant, the influence of the distribution of plugs to households can (unexpectedly) become more relevant than it is in the depicted case, because more groups with less elements have to be considered and delay the processing. Representing such effects is simplified if the model directly represents the behavior of the system.

If we want to use a state of the art modeling approach instead of a manual calculation of the performance, we have to approximate the behavior by deriving the points of time that windows are created, the distribution of their characteristics and partitions, and the delay incurred by grouping operations (i.e., waiting for the end of a window). While we have hinted at how this can be done for simple scenarios where data arrives in regular patterns and where processing does either not take additional time or the additional time can be estimated, it is unclear how this can be done in a systematic way for more complex scenarios.

6 Related Work

There have recently been considerable efforts in modeling the performance of Big Data applications. We identify two main groups of related work.

The first group is approaches that are used to model Big Data systems of different types. Kroß et al. [11, 12] present an approach that utilizes the Palladio modeling language to extract performance models for Apache Spark and Hadoop. Their work is similar in the overall intent, predicting the performance of data-intensive applications. They do not represent stateful operations as model elements that are used in the analysis of the system natively but can represent streaming applications and the performance of streaming frameworks by modeling relevant impact factors, such as number of partitions for a data stream, directly. Other related work models the batch processing of large data sets. For example, Castiglione et al. [4] use an agent-based approach to analyse how highly concurrent big data applications behave in cloud infrastructures regarding the performance, number of used virtual machines and energy efficiency. Aliabadi et al. [1] present an approach that uses Stochastic Activity Networks for modeling different types of batch applications and how they perform when using different frameworks. In the context of the DICE project [3], methods for modeling Big Data systems have been developed. The models explicitly separate between platform, technology and concrete deployment [5] of a big data application. The authors also propose performance simulation methods for simulating their models by transforming models into Petri Nets [6]. Their approach allows modeling complex systems that combine different technologies, including Apache Storm topologies with different types of bolts. We are, however, not aware of an explicit modeling and simulation of stateful operations in their approach. Maddodi et al. [13] present an approach that uses Layered Queuing Networks (LQNs) to analyse the behavior of event-sourcing applications. While they support aggregation of multiple calls for event-sourcing, they do not generalize to other types of aggregation and interaction of calls, such as windows or joins.

The second group of related work addresses systems that process single events but, however, do not target the level of architecture and the abstractions required on this level. Sachs [18] presents an approach for the model-based evaluation of the performance of event-based systems. The work proposes patterns for Queueing Petri Nets that allow architects to model similar behavior as proposed in our approach (such as time windows). However, the work does not target the decomposition of systems on the architecture level. Wu et al. [22] describe a language for defining information needs as queries on event streams and a method for implementing the resulting queries in a high-performance manner. While approaches for specifying complex event processing networks provide similar concepts to the ones presented for our approach, they do not build abstractions that can be used in architecture-level performance models.

Overall, to our knowledge, the state of the art currently does not target the decomposition of data streaming applications in stateful data stream operations and a simulation of this composed system.

7 Conclusion

In this article, we have presented a novel approach for representing and simulating data stream operations in architecture-level component-based performance models. Our approach contributes towards two research questions: 1) how can we represent data stream operations in performance models and 2) how can we analyse the systems' behavior using these models? The representation of data stream operations is relevant, because they commonly appear in an important class of software systems: data-intensive streaming applications. Our evaluation shows that the models of our approach can be used to ease making quantitative statements about the performance of a streaming application because the models express the behavior of the system more accurately, particularly in comparison with a manually derived performance estimate. All of the code of our approach and the artefacts used for this article are publicly available.

Future work will extend our implementation and evaluation and consider the performance impact of employing specific technology realizations for data stream operations. We, furthermore, will investigate the applicability of our approach to other types of systems, particularly self-adaptive systems. Another interesting direction of research is how to extract the type of model presented in this article from code or other artifacts and how to integrate the prediction using these models in an agile software engineering process, as proposed by Mazkatli et al. [14].

References

1. Aliabadi, S.K., et al.: Analytical composite performance models for big data applications. J. Network and Computer Applications **142**, 63–75 (2019)
2. Arasu, A., Babu, S., Widom, J.: The cql continuous query language: semantic foundations and query execution. The VLDB Journal **15**(2), 121–142 (2006)
3. Casale, G., Li, C.: Enhancing big data application design with the DICE framework. In: Advances in Service-Oriented and Cloud Computing – Workshops of ESOCC. pp. 164–168 (2017)
4. Castiglione, A., et al.: Modeling performances of concurrent big data applications. Softw., Pract. Exper. **45**(8), 1127–1144 (2015)
5. DICE consortium: Deliverable 2.4 DICE Deployment Abstractions (2017), http://www.dice-h2020.eu/deliverables/, European Union's Horizon 2020 programme
6. DICE consortium: Deliverable 3.4 DICE simulation tools (2017), http://www.dice-h2020.eu/deliverables/, European Union's Horizon 2020 programme
7. Happe, L., Buhnova, B., Reussner, R.H.: Stateful component-based performance models. Software and Systems Modeling **13**(4), 1319–1343 (2014)
8. Hummel, O., et al.: A Collection of Software Engineering Challenges for Big Data System Development. In: Euromicro SEAA. pp. 362–369. IEEE (2018)
9. Jerzak, Z., Ziekow, H.: DEBS 2014 Grand Challenge: Smart homes – DEBS.org, https://debs.org/grand-challenges/2014/
10. Jerzak, Z., Ziekow, H.: The DEBS 2014 Grand Challenge. In: DEBS '14. pp. 266–269. ACM (2014)
11. Kroß, J., Krcmar, H.: Model-Based Performance Evaluation of Batch and Stream Applications for Big Data. In: MASCOTS. pp. 80–86. IEEE (2017)
12. Kroß, J., Krcmar, H.: Pertract: Model extraction and specification of big data systems for performance prediction by the example of apache spark and hadoop. Big Data Cogn. Comput. **3**(3) (2019)
13. Maddodi, G., Jansen, S., Overeem, M.: Aggregate architecture simulation in event-sourcing applications using layered queuing networks. In: ICPE'20. pp. 238–245. ACM (2020)
14. Mazkatli, M., et al.: Incremental calibration of architectural performance models with parametric dependencies. In: ICSA'20. IEEE (2020)
15. Meijer, E.: Your Mouse is a Database. ACM Queue **10**(3), 20 (2012)
16. Rathfelder, C.: Modelling Event-Based Interactions in Component-Based Architectures for Quantitative System Evaluation. The Karlsruhe Series on Software Design and Quality, KIT Scientific Publishing (2013)
17. Reussner, R.H., et al.: Modeling and Simulating Software Architectures – The Palladio Approach. MIT Press (2016)
18. Sachs, K.: Performance modeling and benchmarking of event-based systems. Ph.D. thesis, Darmstadt University of Technology (2011)
19. Werle, D.: GitHub Repository of Palladio Indirections, https://github.com/PalladioSimulator/Palladio-Addons-Indirections
20. Werle, D.: Data Stream Operations as First-Class Entities in Component-Based Performance Models – Auxiliary Material (2020). https://doi.org/10.5281/zenodo.3937718
21. Werle, D., Seifermann, S., Koziolek, A.: Data stream operations as first-class entities in palladio. In: SSP'19. Softwaretechnik Trends (2019)
22. Wu, E., Diao, Y., Rizvi, S.: High-performance complex event processing over streams. In: SIGMOD. pp. 407–418. ACM (2006)