PROCESS-BASED DATA STREAMING IN SERVICE-ORIENTED ENVIRONMENTS
Application and Technique

Steffen Preissler, Dirk Habich and Wolfgang Lehner
Dresden University of Technology, Database Technology Group, Dresden, Germany

Keywords: Stream, Service, Business process, SOA.

Abstract: Service-oriented environments increasingly become the central point for enterprise-related workflows. This also holds for data-intensive service applications, where such process types encounter performance and resource issues. To tackle these issues on a more conceptual level, we propose a stream-based process execution that is inspired by the typical execution semantics in data management environments. More specifically, we present a data and process model including a generalized concept for stream-based services. In our evaluation, we show that our approach outperforms the execution model of current service-oriented process environments.

1 INTRODUCTION

In order to support analyses and managerial decisions, business people extensively describe the structures and processes of their environment using business process management (BPM) tools (Graml et al., 2007). The area of business processes is well-investigated and existing tools support the life-cycle of business processes from their design, over their execution, to their monitoring today. Business process modeling enables business people to focus on business semantic and to define process flows with graphical support. Prominent business process languages to express such workflows are WSBPEL (OASIS, 2007) and BPMN (OMG, 2009).

The control flow semantic, on which BPM languages and their respective execution engines are based on, has been proven to fit very well for traditional business processes with small-sized data flows. Typical example processes are ”order processing” or ”travel booking”. However, the characteristics of business processes are continuously changing and the complexity grows. One observable trend is the adoption of more application scenarios with more data-intensive processes like business analytics or data integration. Thereby, the volume of data that is processed within a single business process increases significantly (Kouzes et al., 2009).

Figure 1 depicts an example process in the area of business analytics that illustrates the trend to an increased data volume. The process extracts data from different sources and analyzes it in succeeding tasks. First, the process receives a set of customer information as input (`getCustInfos`) which may include customer id, customer name and customer address. Second, the customer ids are extracted and transformed to fit the input structure of both succeeding activities (`transform`). In a third step, the customer ids are enriched concurrently with invoice information (`getInvoices`) and current open orders (`getOrders`) from external services. All open orders are filtered (`filterOrders`) to get only approved orders. Afterwards all information for invoices and orders are joined for every customer id (`joinIDs`) and analyzed (`analyze`). Further activities are executed for different purposes. Since they are not essential for the remainder of the paper, they are denoted by ellipses. The activity type for every task is stated in square brackets ([]) beneath the activity name. These types are derived from BPEL as standard process execution language for Web services.

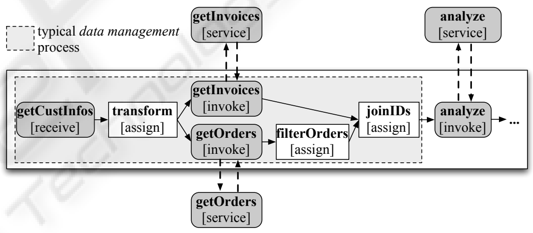


Figure 1: Customer data integration scenario.

As highlighted in Figure 1 by the dotted shaded rectangle, this part of the business process is very similar to typical integration processes within the data management domain with *data extraction*, *data transformation* and *data storage*. In this domain, available modeling and execution concepts for data management tasks are aligned for massive data processing (Habich et al., 2007). Considering the execution concept, Data stream management systems (Abadi et al., 2005) or Extract-Transform-Load (ETL) tools (Vassiliadis et al., 2009) as prominent examples incorporate a completely different execution paradigm. Instead of using a control flow semantic, they utilize data flow concepts with a stream-based semantic that is typically based on *pipeline parallelism*. Furthermore, large data sets are split into smaller subsets. This execution has been proven very successfully for processing large data sets.

Furthermore, many existing work, e.g. (Machado and Ferraz, 2005; Habich et al., 2007; Suzumura et al., 2005), has been demonstrated and evaluated that the SOA execution model is not appropriate for processes with large data sets. Therefore, the changeover from the control flow-based execution model to a stream-based execution model seams essential to react on the changing data characteristics of current business processes. Nevertheless, the key concepts of SOA like flexible orchestration and loosely coupled services have to be preserved. This paper contributes to this restructuring by providing a first integrated approach of a stream-based extension to the service and process level.

Contribution and Outline. Within this paper we make the following contributions: First, we summarize drawbacks of current concepts and present conducted research in the direction of streaming execution in SOA (Section 2). Second, the data flow-based process execution model is presented that allows stream-based data processing (Section 3). Furthermore, our approach for stream-based service invocation in (Preissler et al., 2009) is extended to enable orchestration and usage of web services as streaming data operators (Section 4). Finally, we evaluate our approach in terms of performance (Section 5), present related work (Section 7) and conclude the paper (Section 8).

2 PROBLEM DESCRIPTION

This section consists of two important parts in a survey style. While the first part reviews the current control flow-based execution semantics of today's SOA environments and highlights shortcomings (Section 2.1), the second part briefly describes already conducted work in the direction of streaming semantic in SOA (Section 2.2). There, we summarize open issues which we address in the remainder of the paper.

2.1 Control-flow Semantics in SOAs

The area of business-oriented workflows in SOA is based on the control flow execution of specified *process plans*. Formally, a process plan P is defined as $P = (C, A, S)$, where C is the process context, A is the set of activities that are connected as an acyclic, directed graph and S is the set of services that the process plan interacts with. Furthermore, the process context C is defined as $C = (S_C, V)$ where S_C is a set of system properties like runtime state or process id and V is a set of variables with $V = (v_1, \ldots, v_i, \ldots, v_m)$.

In general, there are two major drawbacks for data-intensive business processes with current SOA concepts. Both are related to control flow-based process execution: (1) on the *process level* with the step-by-step execution model in conjunction with an implicit data flow and (2) on the *service level* with the inefficient, resource consuming communication overhead for data exchange with external services based on the request–response paradigm and XML as data format.

Process Level. Existing process engines execute process plans by applying the specified control flow to every incoming message separately. For n messages, this leads to single process instances P_i with $1 \leq i \leq n$ and $P_i = (C_i, A_i, S_i)$ that are executed serialized in the order of message arrival to preserve data consistency. Every instance is executed as a single thread and the control flow is applied to the message in a step-by-step fashion, where a task is invoked, the response is received and then the next succeeding task is executed in a similar way (Bioernstad et al., 2006).

This single-threaded step-by-step execution in conjunction with the request–response paradigm for service invocation prevents implicit chunking of large data between activities. Thus it also prevents the lowering of peak resource requirements and an increased throughput. In addition, it uses the set V of variables that is located in the process context C for temporal storage and that hides the explicit data flow. Every activity $a \in A$ in process plan P is connected to one ore more variables in V to store outgoing or incoming data for that activity. Figure 2 depicts the architecture of the control flow-based execution with the variable-based data flow between the activities. Thereby, *all* data that is received, transformed and exchanged by every P_i is stored in different variables v_j of V_i.

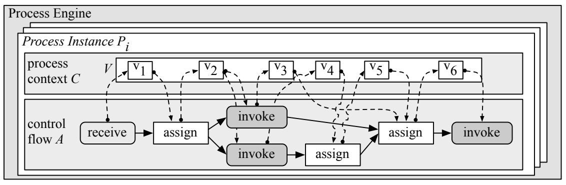


Figure 2: Control flow-based process execution.

Clearly, the implicit, variable-based data flow leads to resource bottlenecks if data structures become large. Considering our application scenario, the efficient execution depends heavily on the number of customer IDs and the number of invoices/orders that are returned for every customer id.

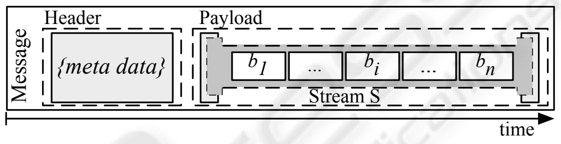
Service Level. On service level, the message-based request-response paradigm between service components in conjunction with XML as data representation poses already well investigated resource problems especially for large and complex structures (Kouzes et al., 2009; Machado and Ferraz, 2005). Quite a lot of research has been proposed that either deals with efficient marshaling and unmarshaling of XML-based content (Suzumura et al., 2005) to reduce communication costs or that considers the partitioning of large messages into smaller chunks (Srivastava et al., 2006; Gounaris et al., 2008) to work around the memory consumption problem. While the former approaches do not consider large message sizes and their in-memory processing, the latter approaches introduce correlation problems when all data that is distributed over many chunks has to be processed in one common context.

The Bottom Line. To conclude this part, the control flow-based execution is not appropriate for data-intensive business processes. The implicit, variable-based data flow and the step-by-step process execution lead to high peak requirements for data storage and decreases throughput. Furthermore, communication overhead between services prevents an efficient implementation and execution of data-intensive business processes. Up to today, services are used as data sources and data sinks in most cases, but the utilization as distributed operators with arbitrary functionality that can be composed with common techniques would extend the applicability of SOA infrastructures to more application domains. Additionally, common business orchestration languages focus on traditional business processes and do not provide native activities for data-related operations like *sort*, *aggregation* or *join*, as e.g. required in our running example.

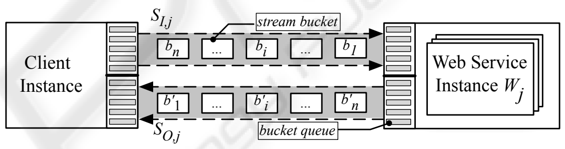
2.2 Streaming Semantics in SOA

In (Preissler et al., 2009), we described the aspect of changing the execution semantic in SOA and introduced the concept of stream-based Web service invocation to overcome the resource restriction with large data sizes. We recall the core concept briefly and point out limitations of this work.

The fundamental idea for the stream-based Web service invocation is to describe the payload of a message as finite stream of equally structured data items. Figure 3(a) depicts the concept in more detail. The message retains as the basic container that



(a) Stream-based message container.


$S_{I,j}$
stream bucket
Client Instance
b_n ... b_i ... b_1
b'_1 ... b'_i ... b'_n
$S_{O,j}$
bucket queue
Web Service Instance W_j

(b) Stream-based message interaction.

Figure 3: Stream-based service invocation.

wraps *header* and *payload* information for requests and responses. However, the payload forms a stream that consists of an arbitrary number n of stream buckets b_i with $1 \leq i \leq n$. Every bucket b_i is an equally structured subset of the application data that usually is an array of sibling elements. Inherently, one common context is defined for all data items that are transferred within the stream. The client controls the insertion of data buckets into the stream and closes the stream on its own behalf. Figure 3(b) depicts the interaction between client and service. Since the concept can be applied bidirectional, a request is defined as input stream $S_{I,j}$ whereas a response is defined as output stream $S_{O,j}$ with j denoting the corresponding service instance. By adding bucket queues to the communication partners, sending and receiving of stream buckets are decoupled from each other (in contrast to the traditional request–response paradigm) and intermediate responses result.

It has been evaluated, that this concept reduces communication overhead in comparison to message chunking by no need for single message creation. In addition, it provides a native common context for all stream items and context sensitive data operations like aggregation can be implemented straightforward. The main drawback of the proposed concept is that it assumes all stream buckets to be application data and equal in structure. This does not take dynamic service parameterization into account. Hence it is not applicable for a more sophisticated stream environment with generalized services operators.

To conclude this section, the stream-based service invocation approach represents only one step in the direction of streaming semantic in service-oriented environments. While the proposed step considers the service level to overcome resource limitations, the process level is obviously an open issue. Therefore, we are going to present a data flow-based process approach for messages processing in the following section. In Section 4, we are extending the service-level streaming technique to cover new arising requirements from the process perspective.

3 STREAM-BASED PROCESS EXECUTION

Fundamentally, our concept for stream-based process execution advances the process level with data flow semantics and introduces a corresponding data and process model for stream-based data processing.

3.1 Data Model

When processing large XML messages, available main memory becomes the bottleneck in most cases. One solution is to split the message payload into smaller subsets and to process them consecutively. This reduces memory peaks by not having to build the whole message payload in memory. To allow native subset processing, we introduce the notion of *processing buckets*, that enclose single message subsets and that are used transparently in the processing framework. Let B be a *process bucket* with $B = (d, t, p_t)$, where d denotes a bucket id, t denotes the bucket type and p_t denotes the XML payload in dependence on the bucket type t. The basic bucket type is *data*, that identifies buckets that contain actual data from message subsets. Of course, a *processing bucket* can also enclose the complete message payload p_m with $p_m == p_t$, as it is the case when the message payload initially enters the process or if p_m is small in size. Nevertheless, for large message payloads it would be beneficial to split them into a set of *process buckets* b_i with $p_{t,i} \subseteq p_m$.

Since buckets carry XML data, XPath and XQuery expressions can be used to query, modify and create the payload structure. As entry point for such expressions, we define two different variables `$_bucket` and `$_system` that define different *access*

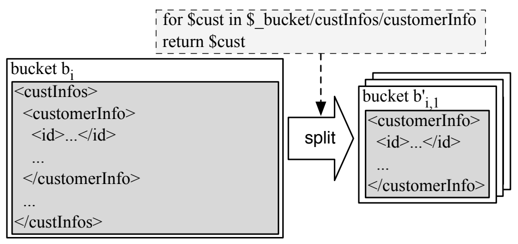


Figure 4: Payload splitting.

paths. Variable `$_bucket` is used to access the bucket payload while variable `$_system` allows access to process-specific variables like process id or runtime state.

Example 1. Payload Splitting. *Consider the activity* `getCustInfos` *from our application scenario in Figure 1. It receives the message with the payload containing a set of customer information. Figure 4 depicts the splitting of this payload into several smaller* process buckets. *The split is described by a very simple XQuery expression with setting the repeating element to* $_bucket/custInfos/customerInfos. *It creates one process bucket for every customer information in the resulting sequence that can be processed consecutively by succeeding activities.*

3.2 Process Model

Instead of using a control flow-based process execution, our process model uses a data flow-based process execution that is based on the *pipes-and-filters* execution model found in various systems like ETL tools, database management systems or data stream management systems. Using the *pipes-and-filters* execution model all activities $a_i \in A$ of a control flow-based process plan P are executed concurrently as independent operators o_i of a pipeline-based process plan P_S. All operators are connected to data queues q_i between the operators that buffer incoming and outgoing data. Hence, a pipeline-based process plan P_S can be described via a directed, acyclic flow graph, where the vertices are operators and the edges between operators are data queues. Figure 5 depicts the execution model of the pipeline-based version of our scenario process. Since the data flow is modeled explicitly, the implicit, variable-based data flow from Figure 2 has been removed. This requires the usage of the additional operator `copy` that copies the incoming bucket for every outgoing data flow.

We define our stream-based process plan P_S as $P_S = (C, O, Q, S)$ with C denoting the process context, O with $O = (o_1, \ldots, o_i, \ldots, o_l)$ denoting the set of operators o_i, Q with $Q = (q_1, \ldots, q_j, \ldots, q_m)$ denoting the set of data queues between the operators and S denoting the set of services the process interacts with. An operator o is defined as $o = (i, o, f, p)$ with i denoting the set of incoming data queues, o denoting the set of outgoing data queues, f denoting the function (or activity type, in reference to traditional workflow languages) that is applied to all incoming data and p denoting the set of parameters that is used to configure f and the operator, respectively.

Figure 6 depicts two succeeding operators o_j and o_{j+1} that are connected by a data queue and that are configured by their parameters p_j and p_{j+1}. Since the operator o_{j+1} processes the data of its predecessor o_j, the payload structure of buckets in queue q_i must match the structure that is expected by operator o_{j+1}. Queues are not conceptually bound to any specific XML structure. This increases the flexibility of data that flows between the operators and can simplify data flow graphs by allowing operators with multiple output structures. Nevertheless, for modeling purposes a set of different XML schemas can be registered to every operator's output that can be used for input validation for the succeeding operators.

Example 2. Schema-free bucket queues. *Consider an XML file containing books and authors as sibling element types. The* `receive` *operator produces buckets with either the schema of books or authors. Since the bucket queues between operators are not conceptually schema-bound, both types can be forwarded directly and, e.g., processed by a* routing *operator that distributes the buckets to different processing flows.*

For parameter set p, we use the respective query languages that where defined with our data model to configure the operator or to retrieve and modify the payload. Clearly, a concrete parameter set of an operator o is solely defined by function f. For f, we define a set of predefined algorithms that are needed for sophisticated data processing. Inspired by (Böhm et al., 2009), we define three classes of basic functions that semantically provide a foundation for data processing: These classes are *interaction-oriented functions* including `invoke`, `receive` and `reply`, *control-flow-oriented functions* including `route`, `copy`, and `signal` and *data-flow-oriented functions* including `assign`, `split`, `join`, `union`, `orderby`, `groupby`, `sort` and `filter`. All functions work on the granularity of *process bucket B* and are implemented as

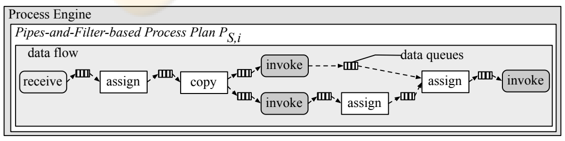


Figure 5: Pipeline-based execution of process plan P_S.

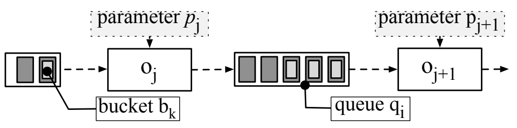


Figure 6: Operators and *Processing Bucket* queue.

operators.

Now, we discuss the semantics of example operators for every function class in more detail. In particular, this will be *receive*, for the class of *interaction-oriented functions*, *copy*, for the class of *control-flow-oriented functions*, and *join*, for the class of *data-flow-oriented functions*.

Receive Operator. The most important operator for preparing incoming messages is the `receive` operator. This operator gets one bucket with the payload of the incoming message to process. As parameter p, a split expression in XPath or XQuery must be specified to create new buckets for every item in the resulting sequence. It is closely related to the `split` operator. While `split` is used within the data flow to subdivide buckets, receive is linked to the incoming messages and usually starts the process. Please, refer to Example 1 for the usage of the `receive` operator.

Copy Operator. The copy operator, as it is used in our application scenario, has one input queue and multiple output queues. It is used to execute concurrent data flows with the same data. For l output queues it creates $l - 1$ copies of every input bucket and feeds them all output queues.

Join Operator. The join operator can have different semantics and usually joins two incoming streams of buckets. For this paper, we describe an equi-join that is implemented as a *sort-merge* join. This requires the join keys to be ordered. In our application scenario, this ordering is given inherently by the data set. Alternatively, the *receive* operator can be parameterized to order all items by a certain key. If this requirement cannot be fulfilled, another join algorithm has to be chosen. The set of parameters for a join operator includes (1) paths to both input bucket stream elements, (2) the paths to both key values that have to be equal and (3) the paths to the target destination in the output structures of the join operator.

Example 3. Merge Join Operator. *Figure 7 depicts the join operator* `joinIDs` *that joins the payload of order buckets and invoice buckets into one bucket for each customer id. Thus, the customer id attribute is the join key in both input streams. The join key paths are denoted by* `$left_key` *and*

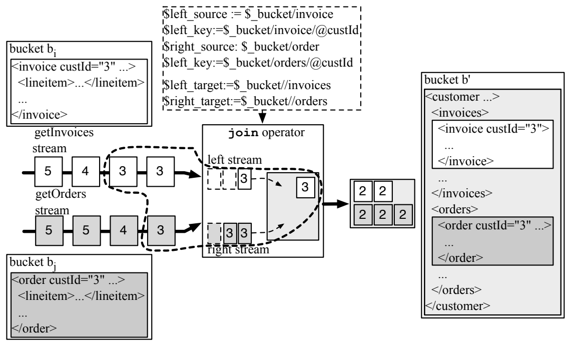


Figure 7: Join operator.

`$right_key`. Although the invocation of stream-based services will be discussed in the next section, assume that the `getInvoices` operator produces one bucket for every invoice per customer id. Thus, buckets with equal customer id arrive in a grouped fashion due to the preceding invoke operators. The join algorithm takes every incoming bucket from both input streams and compares the id that is currently joined. In our example, the current id is 3. If the bucket ids equal the current id, the payloads according `$left_source` and `$right_source` are extracted from that buckets and inserted into the new output bucket according the target paths `$right_target` and `$right_target`. If the ids of both streams become unequal, the created bucket is passed to the succeeding operator as it is already done for id 2.

4 GENERALIZED STREAM-BASED SERVICES

Taking the presented process execution as our foundation, we address the communication between process and services in this section. The general idea is to develop stream-based services that operate 1) as efficient, stream-based services for traditional service operations like data extraction and data storage and 2) as stream operators for data-oriented functionalities and for data analysis. This enables our stream-based process model to integrate and orchestrate such services natively as remote operators. Thus, the process can decide whether to execute an operator locally or in distributed fashion on a different network node.

4.1 Service Invocation Extension

The main drawback of the presented stream-based service invocation approach from Section 2.2 is the missing support for service parameterization. Only raw application data and thus only one data structure without metadata is supported. Parameters are not considered specifically and the only way to pass parameters to the service is to incorporate them into the application data structure (see Figure 8a). This blurs the semantics of both distinct structure types and creates overhead if the parameter only initializes the service instance. Furthermore, a mapping between stream buckets with its blurred structure on the service level and our processing buckets on the process level has to be applied.

Example 4. Drawback of Single Bucket Structure. *In our application scenario, the parameter for the service `getInvoices` would be a time frame definition, in which all returned invoices had to be created. Although this one time frame is valid for all customer ids that are processed by the service, it has to be transmitted with every stream bucket.*

We extend the stream item definition by deploying the proposed *process bucket* definition B from our *data model* directly into the invocation stream. Remember, a process bucket is described by its *type* t and the payload p_t that depends on t. Hence, we denote buckets that carry application data with $t = data$. We introduce parameter buckets for service initialization or reconfiguration by adding a new type t with $t = param$. Thus, parameters are separate buckets that have their own payload and that are processed by the service differently. Figure 8b depicts the concept of parameter separation. Besides a more clear separation, this concept generalizes stream-based service implementations by allowing to deploy parameterizable functions as Web services that are executed on stream buckets.

Furthermore, it enables us to incorporate these services as remote operators into our process model. A parameter set p that is currently used to configure a local operator o_i can be transferred to the service via dedicated parameter buckets. Hence, this service can act as a remote operator, if it implements the same function f. The conceptual distinction between remote service and local operator becomes almost negligible.

Example 5. Generalized Filter Service. *Consider the `filterOrders` operator in our application scenario. It filters incoming order buckets according to the order's status. The filter expression is described*

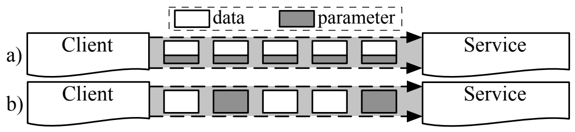


Figure 8: Extended bucket concept.

as XPath statement in its parameter set p. If the filter algorithm f is deployed as a Web service, it is configured with p using the parameter bucket structure. Thus, a service instance can filter arbitrary XML content according to the currently configured filter expression.

Of course, central execution will certainly dominate the communication overhead compared to a distributed execution. But further research should investigate this in more detail.

4.2 Classification and Applicability

In order to integrate stream-based services as data sources and as remote operators into our process execution, we first have to classify our defined process operators according incoming and outgoing data flows. In a second step, we investigate how to map these operator classes to stream-based services. Following (Vassiliadis et al., 2009), we can classify most operators into *unary* operators (one input edge, one output edge, e.g.: `invoke`, `signal` and `groupby`) and *binary* operators (two input edges, one output edge, e.g.: `join` and `union`). Furthermore, *unary* operators can have an input–output relationship of *1:1*, *1:N* and *N:1*.

Applicability as unary operator: Naturally, a stream-based service has one input stream and one output stream. Therefore, it can be directly mapped to an *unary* operator. Since the receiving and sending of *process buckets* within a service instance are decoupled, the input–output relationships of *1:1*, *1:N* and *N:1* are supported in straightforward fashion.

Example 6. $1 : N$ Relationship. *Consider our data source* `getInvoices`*. The corresponding invoke operator is depicted in Figure 9.First, the service is configured using the parameter set with* `$valid_year=2009` *as predicate, so that only invoices that were created in 2009 will be returned. Second, since the service directly accepts process buckets, input buckets containing single customer ids are streamed to the service. These customer id buckets are the result of the split in the* `getCustInfos` *operator and the succeeding* `transform` *operator. The service retrieves all invoices and returns every invoice for every customer id in one separate response bucket. Hence, the presented service realizes a* $1 : N$ *input-output relationship. Since the service returns* process buckets*, they can be directly forwarded to the* `joinIDs` *operator.*

Applicability as binary operator: Since a stream-based service typically provides only one input

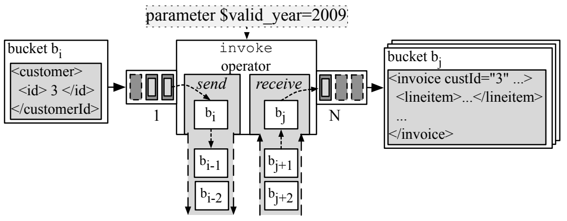


Figure 9: Invoke operator.

stream, it cannot be mapped directly to *binary* operators. A simple approach to map the stream-based service to the type of a *binary* operator is to place all buckets from both input queues to the one request stream to the service and to let the service validate which bucket belongs to which operator input. As a first step, we focus on this approach and also implemented it for our evaluation in Section 5. Further research should investigate if a more sophisticated approach, e.g., one that implements two concurrent streams for one service instance, would be more applicable.

5 EVALUATION

In this section, we provide performance measurements for our stream-based process execution. In general, it can be stated that the stream-based message processing leads to significant performance improvements and scales for different data sizes.

5.1 Experimental Setup

We implemented our concept using Java 1.6 and the Web service framework Axis2[1], and we ran our process instances on a standard blade with 3 GB Ram and four cores at 2 GHz. The data sources were hosted on a dual core workstation with 2 GB Ram connected in a LAN environment. Both nodes were assigned 1.5 GB Ram as Java Heap Size. All experiments were executed on synthetically generated XML data and were repeated 30 times for statistical correctness.

We used our running process example. For the traditional process execution, the process graph consists of seven nodes: one receive activity, three *assign* activities (`transform`, `filterOrders` and `joinIds`), and three *invoke* activities (`getInvoices`, `getOrders` and `analyze`). For our stream-based process execution the process graph consists of eight nodes. We additionally have the *copy* operator that distributes the customer ids to both *invoke* operators. Furthermore,

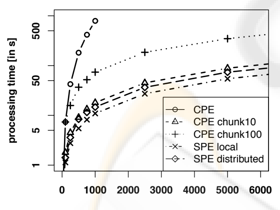
[1]http://ws.apache.org/axis2/

we replace the *assign* operators `filterOrders` and `joinIds` with the respective *filter* and *join* operator.

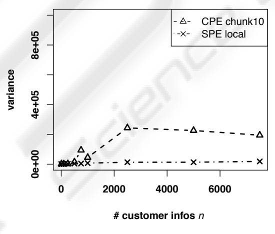
We use *n* as the number of customer information that enter the process. In addition, we fix the number fn of invoices and orders returned for each customer id from the services `getInvoices` and `getOrders` to 10 for all conducted experiments. This leads to 20 invoices/orders for every processed customer id. The textual representation of one customer information item that enters the process is about 1kb in size. It gets transformed, enriched and joined to about 64kb throughout the process. Although real-world scenarios for data integration often exhibit larger message sizes, these sizes are sufficient for comparing the presented approaches.

5.2 Performance Measurements

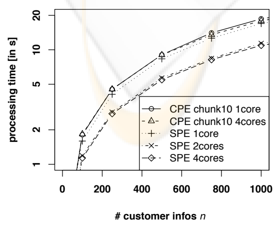
For scalability over *n*, we measured the processing time for the traditional control-flow-based process execution (*CPE*) and our stream-based process execution (*SPE*) in Figure 10(a) with a logarithmic scale. Thereby, *CPE* denotes the control-flow-based execution which processes all customer information *n* in one process instance. *CPE chunk10* and *CPE chunk100* uses the CPE but distribute *n* items over $n/chunkSize$ service calls with $chunkSize = \{10, 100\}$. *SPE local* represents our stream-based process execution with only `getInvoices`, `getOrders` and `analyze` being stream-based invoke operations to external Web services. In contrast, *SPE distributed* replaces the join operator `joinIDs` with a binary invoke operator described in Section 4.2 and implements the join as stream-based service instance.



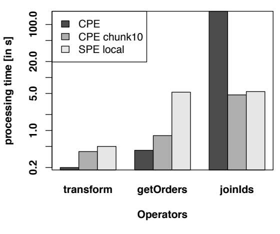
(a) Scalability Over *n*.



(b) Variance over *n*.



(c) Influence CPU cores.



(d) Execution Times.

Figure 10: Experimental performance evaluation results.

We can observe, that *CPE* does not scale over 1.000 customer ids with its 20.000 invoices/orders due to main memory limit of 1.5 GB and its variable-based data flow which stores all data. In contrast, *CPE chunk10* and *CPE chunk100* scale for arbitrary data sizes whereas a chunk size of 10 customer information per process call offers the shortest processing time. Nevertheless, this data chunking leads to multiple process calls for a specific *n* and alters the processing semantic by executing each process call in an isolated context and thus assuming independency between all *n* items. Furthermore, it also exhibits a more worse runtime behavior than *SPE local* and *SPE distributed* Since chunking scales for arbitrary data sizes, we will focus on these approaches in our following experiments.

Figure 10(b) depicts the variance of runtimes for *CPE chunk10*, *CPE chunk100* and *SPE local*. While the variance of both chunk-based control-flow execution concepts offers a higher variance for larger *n*, the variance of our stream-based execution shows significantly lower. This is due to the fact, that with higher *n* the number of service calls increases for the *CPE* approaches, which also involves service instance creation and the all new routing of the message the service endpoint. In contrast, our stream-based service invocation only creates one service instance per invoke operator and the established streams are kept open for all *n* that flow through the process.

In Figure 10(c) we measured the runtime performance with different numbers of dedicated CPU cores to the process instances. We can observe, that the number of cores does not affect the *CPE* approaches significantly. This is due to the fact that at most 2 threads are executed concurrently (both concurrent invokes for `getInvoices` and `getOrders` in the process graph). Furthermore, waiting times for the return of the service calls (processing, creation and transmission of invoices and orders) does even not fully utilize one core and makes the presence of the remaining three cores obsolete. For the *SPE* approach, we have 12 threads (8 operator nodes with one thread per operator plus an additional thread for every *invoke* (+3) and *join* (+1) operator). The execution time for different *n* is significantly higher with only using one CPU core. Nevertheless, the *SPE* also outperforms the *CPE* with one single CPU core. Again this points to the fact of dominating waiting times for service calls in *CPE*-based processes. The assignment of two CPU cores speeds up the SPE significantly whereas 4 CPU cores do not increase performance that may justify its dedicated usage.

As a last experiment, Figure 10(d) depicts execution times for different operators of *CPE*, *CPE* *chunk10* and *SPE local* in a logarithmic scale. Here we fix $n = 500$ and measured the time the operators finish to process all items. In general, the execution of *SPE* operators takes more time than the corresponding activities in the *CPE* environment. This is due to overhead for operator scheduling, queue synchronization and bucket handling. Furthermore, the operator `getOrders` has nearly the same execution time like its succeeding operator `joinIDs` for *SPE local*. This points to the fact, that the `getOrders` operator is the most time consuming operator and `joinIDs` operator has to wait for buckets. As for the `joinIDs` activity in the `CPE`, is the most time and resource consuming step. We used a standard Java XPath library to implement the join for the *CPE*. Thereby, all invoices and orders for every customer id are retrieved from variable $v3$ and $v5$ (see Figure 2) and stored in variable $v6$. For our *SPE* implementation, we used the same library and algorithm but process only small message subsets. This seemed to speed up the whole data processing significantly.

6 DISCUSSION

In general, the evaluation in Section 5 has shown, that the pipeline-based execution in conjunction with the stream-based Web service invocation yields significant performance and scalability improvements for our application scenario. In the following section we discuss implications and challenges for our process-based data streaming approach. In particular, there are three topics that consider different aspects of our approach:

Applicability and Application Scenarios. In our presented application scenario, we considered equally structured items.This is a typical data characteristic in data-intensive processes. However, if items are not equally-structured, our process model supports this by schema-less data queues (see Example 2). Another quite interesting application domain is the process-based inter-message processing in the area of message stream analysis. It has to be investigated, how decision rules can be mapped to process graph and how Complex Event Processing (CEP) can be embedded in this context.

Intra-process Optimization. Since our approach allows for data splitting, it scales for arbitrary data sizes. However, the scalability depends on the bucket payload size and the number of buckets within the process. Since all data queues block access if they become empty or full, the maximum number of buckets within the process is implicitly defined by the sum of slots in all data queues. Furthermore, if a customer id and its invoices/orders are processed completely, their buckets are consumed by the `analyze` operator and discarded afterwards. Another implication in terms of bucket payload sizes is that the `receive` operator does not build the incoming message payload in memory completely. Instead, it reads the message payload from an internal storage and parses the XML file step by step, according the split path in the `receive` operator. Of course, this may restrict the expressiveness of XQuery statements.

Inter-process Optimization. Currently, the pipeline-based execution is only deployed on an intra-process-based level. Thereby, every incoming message is processed in a pipeline-based, but different incoming messages are executed in separate instances in a serialized fashion. One optimization would be to allow new messages to be processed in the same instance as the previous messages. This leads to the processing of a new message while the previous message is still be processed. However, the process has to distinguish between single messages to maintain separate contexts. To mark the end of an old message and the start of new message, respectively, we can use punctuation buckets as described in (Tucker et al., 2003) that are injected into the stream between two messages. In these bucket types, message meta data like request id or response endpoints are included. Additionally, the process model has to be extended to allow punctuations to reconfigure operators via parameter sets for every message. If punctuations are not used at all, payloads from different messages can be processed in one shared context which allows new application scenarios in the area of message stream analysis.

7 RELATED WORK

In general, there exist several papers addressing the optimization of business processes. The closest related work to ours is (Bioernstad et al., 2006). They investigate runtime states of activities and their pipelined execution semantics. However, their proposal is more a theoretical consideration with regard to single activities. They describe how activities have to be adjusted to enable pipelined processing, whereas they do not consider 1) message splitting for efficient message processing and 2) communication with external systems. Furthermore, (Böhm et al., 2009) addresses the transparent rewriting of instance-based processes to pipeline-based processes by considering cost-based rewriting rules. Similar to (Bioernstad et al., 2006), they do not address the optimization of data-intensive processes.

The optimization of data-intensive business processes is investigated in (Maier et al., 2005; Vrhovnik et al., 2007) and (Habich et al., 2007). While (Maier et al., 2005) proposes to extend WSBPEL with explicit database activities (SQL-statements), (Vrhovnik et al., 2007) describes optimization techniques for such SQL-aware business processes. In contrast to our work, their focus is on database operations in tight combination with business processes. (Habich et al., 2007) presents an overall service-oriented solution for data-intensive applications that handles the data flow separately from process execution and uses database systems and specialized data propagation tools for data exchange. However, the execution semantics of business processes is not touched and only the data flow is optimized with special concepts – restricting the general usability of this approach in a wider range.

8 CONCLUSIONS

In this paper we presented the concept of stream-based XML data processing in SOA using common service-oriented concepts and techniques. There, we used pipeline parallelism to process data in smaller chunks. In addition, we addressed the communication between process and services and introduced the concept of generalized stream-based services that allows the process to execute services as distributed operators. In experiments we showed the applicability of these concepts in terms of performance. Future work should address 1) the modeling aspects of such processes in more detail and 2) a cost model that considers communication overhead, complexity of functions, and complexity and size of data structures to determine the remote or local execution of operators.

REFERENCES

Abadi, D. J., Ahmad, Y., Balazinska, M., Cetintemel, U., Cherniack, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S. (2005). The Design of the Borealis Stream Processing Engine. In *CIDR*.

Bioernstad, B., Pautasso, C., and Alonso, G. (2006). Control the flow: How to safely compose streaming services into business processes. In *IEEE SCC*.

Böhm, M., Habich, D., Preissler, S., Lehner, W., and Wloka, U. (2009). Cost-based vectorization of instance-based integration processes. In *ADBIS*.

Gounaris, A., Yfoulis, C., Sakellariou, R., and Dikaiakos, M. D. (2008). Robust runtime optimization of data transfer in queries over web services. In *ICDE*.

Graml, T., Bracht, R., and Spies, M. (2007). Patterns of business rules to enable agile business processes. In *EDOC*.

Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W., and Maier, A. (2007). Bpel-dt - data-aware extension of bpel to support data-intensive service applications. In *WEWST*.

Kouzes, R. T., Anderson, G. A., Elbert, S. T., Gorton, I., and Gracio, D. K. (2009). The changing paradigm of data-intensive computing. *IEEE Computer*.

Machado, A. C. C. and Ferraz, C. A. G. (2005). Guidelines for performance evaluation of web services. In *WebMedia*.

Maier, A., Mitschang, B., Leymann, F., and Wolfson, D. (2005). On combining business process integration and etl technologies. In *BTW*.

OASIS (2007). Web services business process execution language 2.0 (ws-bpel). `http://www.oasis-open.org/committees/wsbpel/`.

OMG (2009). Business process modeling language 1.2. `http://www.omg.org/spec/BPMN/1.2/PDF/`.

Preissler, S., Voigt, H., Habich, D., and Lehner, W. (2009). Stream-based web service invocation. In *BTW*.

Srivastava, U., Munagala, K., Widom, J., and Motwani, R. (2006). Query optimization over web services. In *VLDB*.

Suzumura, T., Takase, T., and Tatsubori, M. (2005). Optimizing web services performance by differential deserialization. In *ICWS*.

Tucker, P. A., Maier, D., Sheard, T., and Fegaras, L. (2003). Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568.

Vassiliadis, P., Simitsis, A., and Baikousi, E. (2009). A taxonomy of etl activities. In *DOLAP*.

Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., and Kraft, T. (2007). An approach to optimize data processing in business processes. In *VLDB*.