Steffen Preissler, Dirk Habich, Wolfgang Lehner

An XML-Based Streaming Concept for Business Process Execution

Erstveröffentlichung in / First published in:

Enterprise Information Systems: 12th International Conference. Funchal-Madeira, 08.-12.06.2010. Springer, S. 60-75. ISBN 978-3-642-19802-1.

DOI: http://dx.doi.org/10.1007/978-3-642-19802-1_5

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5

An XML-Based Streaming Concept for Business Process Execution

Steffen Preissler, Dirk Habich, and Wolfgang Lehner

Dresden University of Technology, Dresden 01187, Germany
{steffen.preissler,dirk.habich,wolfgang.lehner}@tu-dresden.de
http://wwwdb.inf.tu-dresden.de/~research

Abstract. Service-oriented environments are central backbone of todays enterprise workflows. These workflow includes traditional process types like travel booking or order processing as well as data-intensive integration processes like operational business intelligence and data analytics. For the latter process types, current execution semantics and concepts do not scale very well in terms of performance and resource consumption. In this paper, we present a concept for data streaming in business processes that is inspired by the typical execution semantics in data management environments. Therefore, we present a conceptual process and execution model that leverages the idea of stream-based service invocation for a scalable and efficient process execution. In selected results of the evaluation we show, that it outperforms the execution model of current process engines.

Keywords: Stream, Service, Business process, SOA.

1 Introduction

In order to support managerial decisions in enterprise workflows, business people describe the structures and processes of their environment using business process management (BPM) tools [1]. 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 are WSBPEL [2] and BPMN [3].

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 [4].

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

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5

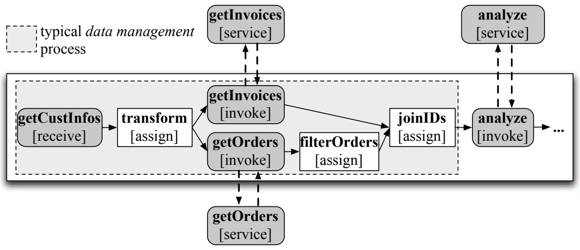


Fig. 1. Customer Data Integration Scenario

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.

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 [5]. Considering the execution concept, Data stream management systems [6] or Extract-Transform-Load (ETL) tools [7] 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. [8,5,9], 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. In this paper we contribute as follows: First, we summarize give a brief introduction into the concept of stream-based service invocation 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 [10] is extended to enable orchestration and usage of web services as streaming data operators (Section 4). In Section 5, we discuss optimizations for our execution concept. Finally, we evaluate our approach in terms of performance (Section 6), present related work (Section 7) and conclude the paper (Section 8).

2 Stream-Based Service Invocation Revisited

In [11], we describe the concepts of control flow-based process execution that are used in today's SOA environments and highlight it's shortcomings in terms of data-intensive service applications. In a nutshell, two major drawbacks for data-intensive business processes have been identified. 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.

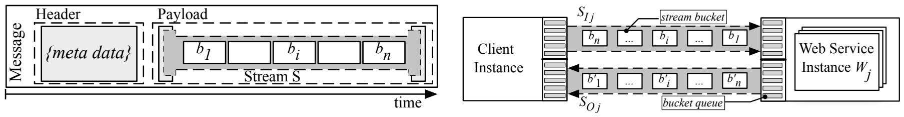
In [10], we already tackled the *service level* aspect by introducing the concept of stream-based Web service invocation to overcome the resource restriction with large data sizes. We recall the core concept briefly and point out limitations of this work.

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 2(a) depicts the concept in more detail. The message retains as the basic container that 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 2(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* aspect to overcome resource limitations, the *process level* aspect 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 requirements from the proposed process perspective.

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5



(a) Stream-Based Message Container. (b) Stream-Based Message Interaction.

Fig. 2. Stream-based Service Invocation

3 Stream-Based Process Execution

Basically, 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 *process 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 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.

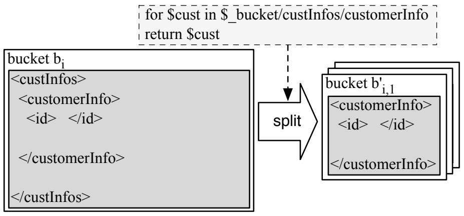


Fig. 3. Payload Splitting

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 3 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 4 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 the traditional instance-based execution as described in [11] 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 5 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.

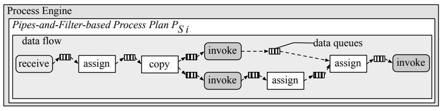


Fig. 4. Pipeline-Based Execution of Process Plan P_S

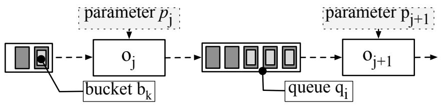


Fig. 5. Operators and *Processing Bucket* Queue

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 [12], 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 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.

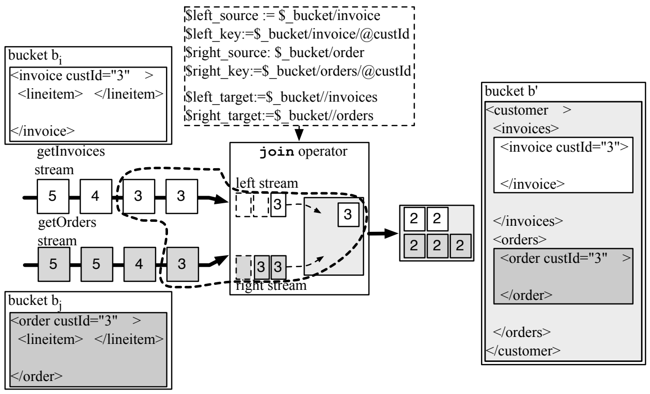


Fig. 6. Join Operator

Example 3. Merge join operator: Figure 6 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 `$left_key` and `$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 `$left_target` and `$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 is the missing support for service parameterization. Only raw application data

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5

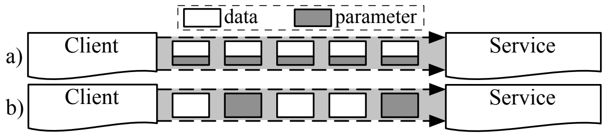


Fig. 7. Extended Bucket Concept

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 7a). 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 7b 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. Of course, central execution will certainly dominate the communication overhead compared to a distributed execution. But further research should investigate this in more detail.

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 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.

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 [7], 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 8. 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 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 6. 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.

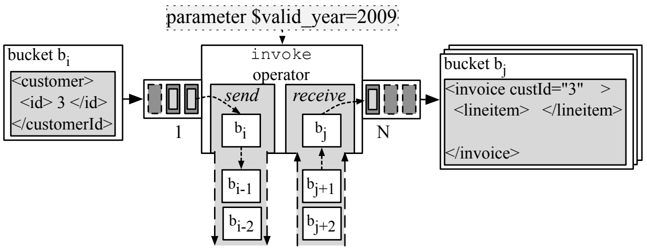


Fig. 8. Invoke Operator

5 Process Model Optimizations and Extensions

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:

5.1 Process Execution Optimization

Our optimization considerations can be classified into two main sections: (1) intra-process optimizations, which analyzes optimization possibilities within one process instance and (2) inter-process optimizations, which analyzes performance improvements between consecutive process executions.

Intra-process Consideration and Optimization. Since our approach allows for data splitting, it scales for arbitrary data sizes. However one implication is, that the scalability depends on the bucket payload size and the number of buckets within the process. Since all data queues block new insertions 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. Therefore we consider different queue sizes between the operators as possible optimization parameter where processes with small queue sizes will require less main memory but will be more prone to communication latencies for single buckets. Instead processes with larger operator queues consume more main memory but will be able to terminate fast preceding operators and free resources, while slow succeeding operators in the chain can process their items more slowly without slowing down a fast preceding one.

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, since the processing is forward only. Furthermore the payload size depends on the XPath expression and the input data structure. An optimization technique in this area is the packaging of single bucket payloads into one physical bucket. While the logical separation of each payload is preserved by the operators via modifications in the operator's functionality, there are much less physical buckets in the systems. We will analyze whether, and if, how many payloads have to be packed into one physical bucket to improve process execution time.

Inter-process Optimization. Currently, the pipeline-based execution is only deployed on an intra-process-based level. Thereby, the payload of every incoming message is processed in pipeline-based, but different incoming messages are executed in separate instances in consecutive executions. One optimization is 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 start of a new message and thus the end of an old message, we deploy punctuation as described in [13] by extending the *data model* and introducing a new bucket type

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5

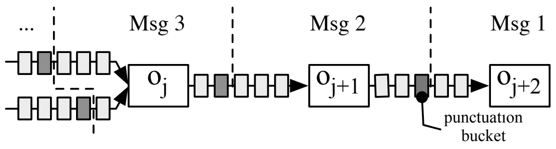


Fig. 9. Inter-process Optimization with Punctuation Buckets

$t = SEPCTX$. Such a punctuation bucket is injected into the stream when a new message starts (see Figure 9). It has a predefined payload structure in p_t which includes meta data from the next message like request id or response endpoint that are needed to communicate results appropriately.

Furthermore, the *process model* is extended to be aware of punctuations and to ensure the separation of message contexts. In general, if an operator consume a punctuation bucket from the input queue, it resets its internal state and forwards the punctuation bucket to each outgoing queue. For the `copy` operator, this implies that it copies the punctuation bucket for every outgoing queue. Special considerations have to be made for binary operators like `join` or `union` and for the `invoke` operator. Since binary operators merge concurrent data flows, the operator function has to stop consuming from one queue, if it encounters the punctuation in that queue. The operator has to consume from the second queue, until it also encounters the punctuation bucket in the second queue. Then, both punctuations are removed from the queue and one of them is placed in the outgoing queue. For the `invoke` activity are two possibilities exist for punctuation handling. If the remote service does not support punctuations, the operator has to shut down the invocation stream normally as it would be the case when shutting down the whole process normally. Afterwards, the punctuation is placed in the outgoing queue and the new invocation stream is established for the next message context. If the service supports punctuations, the operator has not be aware of any punctuation semantic. The invocation stream is kept open and the punctuation is placed in the stream with all other buckets, whereas the service resets its internal state and forwards the punctuation back to the operator. Both types have been implemented and are evaluated in Section 6.

If punctuations are not used at all but different message payloads are processed in one shared context, it allows new application scenarios in the area of message stream analysis that is described next.

5.2 Applicability and Operator Extension

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) and multiple definitions in the `route` operator. Another reasonable application domain is inter-message processing in the area of message stream analysis. Consider services, that are monitored via sensors. These sensors forward metrics like response time and availability in a predefined time interval. Decision rules and action chains can be defined as a process-based data streaming application and would enable the reduction of orthogonal, not XML-based systems like traditional data stream management systems. To support such scenarios, we can leverage the semantics of punctuations described earlier to provide one process instance and thus one common context for all arriving messages. Additionally, only the process model has to be extended with more operator types to support event processing operations like *time windows* and *sequences*. Furthermore, it has to be investigated, how decision rules can be mapped to our process graph and how Complex Event Processing (CEP) can be embedded in this context.

6 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.

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

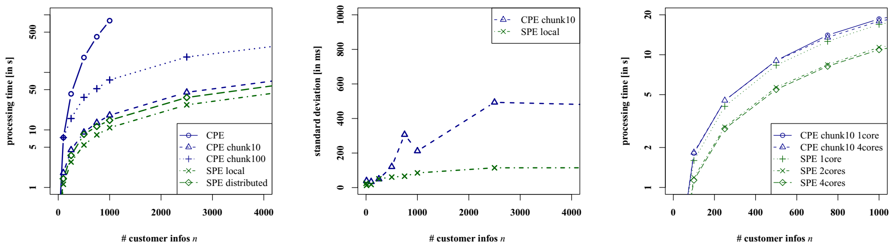
We use n as the number of customer information that enter the process. In addition, we fix the number of invoices and orders returned for each customer id from the services `getInvoices` and `getOrders` to 10 for all conducted experiments. This leads to 20 invoices/orders for every processed customer 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.

6.2 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*

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

Final edited form was published in "Enterprise Information Systems: 12th International Conference. Funchal-Madeira 2010", S. 60-75, ISBN 978-3-642-19802-1
http://dx.doi.org/10.1007/978-3-642-19802-1_5



(a) Scalability Over n. (b) Standard deviation over n. (c) Influence CPU cores.

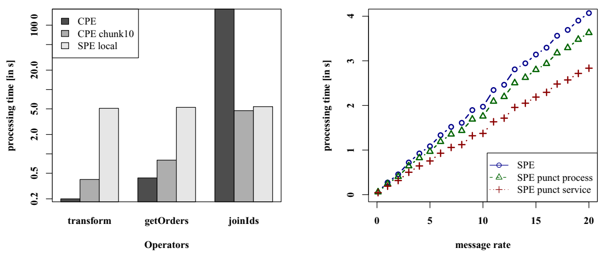
Fig. 10. Experimental Performance Evaluation Results

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.

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 standard deviation of runtimes for *CPE chunk10*, *CPE chunk100* and *SPE local*. While the standard deviation of both chunk-based control-flow execution concepts increases for larger n, the standard deviation 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 to 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 it.

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



(a) Operator Execution Times. (b) SPE Optimizations.

Fig. 11. Experimental Performance Evaluation Results

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 fourth experiment, Figure 11(a) 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 every *SPE* operator takes more time than the corresponding activities in the *CPE* environment (except the CPE invoke). Furthermore, all SPE operators run quite the same amount of time. This is due to the pipelined execution of all operators with its blocking queue semantic. So while the `transformID` operator starts running, all succeeding operators are also started appropriately. So while the `transformID` operator processes further buckets, e.g. the `joinIDs` operator already processes buckets from it's input queues. The blocking nature of the operators implies, that the pipeline is only as fast as the slowest operator in the chain. 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 internal variables (see [11] for more detail) and stored in internal variables. For our SPE implementation, we used the same library and algorithm but process only small message subsets with each join step. This seemed to speed up the whole data processing significantly.

In Figure 11(b), we focused solely on our *SPE* implementation and compared the plain execution with the optimized execution considering the punctuation semantics from Section 5. Therefore, we fixed the message size to $n = 10$ and varied the message rate r from 1 to 20. We measured cumulative time to finish all messages. Thereby, *SPE* denotes the plain execution without optimization. Furthermore, *SPE punct process* denotes the optimization where a punctuation is not supported by services and thus terminates the invocation stream. Finally, *SPE punct service* denotes the optimization where punctuations are supported by the stream-based services and thus the invocation streams can be kept open. As expected, the plain *SPE* implementation is the most inefficient for consecutive process execution. While it scales very good with increasing data volume *within* one process instance, it becomes a bottleneck if many consecutive instances have to be created. This includes the instantiation of every operator before any processing can be done and the expensive invocation stream establishment for every `invoke` operator. On the opposite, *SPE punct process* keeps all operators and the current process instance for consecutive requests. Thus, it eliminates operator creation overhead and scales much better with increasing message rate. Finally, *SPE punct service* additionally eliminates the expensive network stream establishment for consecutive requests and the whole process pipeline remains filled.

To conclude, the evaluation of our concept 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.

7 Related Work

In general, there exist several papers addressing the optimization of business processes. The closest related work to ours is [14]. 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, [12] addresses the transparent rewriting of instance-based processes to pipeline-based processes by considering cost-based rewriting rules. Similar to [14], they do not address the optimization of data-intensive processes.

The optimization of data-intensive business processes is investigated in [15,16] and [5]. While [15] proposes to extend WSBPEL with explicit database activities (SQL-statements), [16] 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. [5] 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 pieces. In addition, we addressed the communication between process and services and introduced the concept of generalized stream-based services. It allows the process to execute services as distributed operators with arbitrary functionality. Furthermore, we presented optimizations to increase scalability for inter-process message processing. In experiments we showed the applicability of these concepts in terms of performance. Future work should address the modeling aspects of such processes in more detail. More specifically, it should be investigated, how notations like BPMN in conjunction with annotated business rules have to be mapped to an operator graph of our process model. Additionally, a cost model is reasonable that considers communication costs, complexity of data structures and complexity of implemented operator functions to advise the remote or local execution of process operators.

References

1. Graml, T., Bracht, R., Spies, M.: Patterns of business rules to enable agile business processes. Enterp. Inf. Syst. 2(4), 385–402 (2008)
2. OASIS. Web services business process execution language 2.0 (ws-bpel) (2007), `http://www.oasis-open.org/committees/wsbpel/`
3. OMG. Business process modeling language 1.2 (2009), `http://www.omg.org/spec/BPMN/1.2/PDF/`
4. Kouzes, R.T., Anderson, G.A., Elbert, S.T., Gorton, I., Gracio, D.K.: The changing paradigm of data-intensive computing. IEEE Computer (2009)
5. Habich, D., Richly, S., Preissler, S., Grasselt, M., Lehner, W., Maier, A.: Bpel-dt - data-aware extension of bpel to support data-intensive service applications. In: WEWST (2007)
6. 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., Zdonik, S.: The Design of the Borealis Stream Processing Engine. In: CIDR (2005)
7. Vassiliadis, P., Simitsis, A., Baikousi, E.: A taxonomy of etl activities. In: DOLAP (2009)
8. Machado, A.C.C., Ferraz, C.A.G.: Guidelines for performance evaluation of web services. In: Proceedings of the 11th Brazilian Symposium on Multimedia and the Web, WebMedia 2005, pp. 1–10. ACM Press, New York (2005)
9. Suzumura, T., Takase, T., Tatsubori, M.: Optimizing web services performance by differential deserialization. In: Proceedings of the 2005 IEEE International Conference on Web Services, ICWS 2005, Orlando, FL, USA, July 11-15, pp. 185–192 (2005)
10. Preissler, S., Voigt, H., Habich, D., Lehner, W.: Stream-based web service invocation. In: BTW (2009)
11. Preissler, S., Habich, D., Lehner, W.: Process-based data streaming in service-oriented environments - application and technique. In: Filipe, J., Cordeiro, J. (eds.) ICEIS 2010. LNBIP, vol. 73, pp. 56–71. Springer, Heidelberg (2011)
12. Boehm, M., Habich, D., Preissler, S., Lehner, W., Wloka, U.: Cost-based vectorization of instance-based integration processes. In: Grundspenkis, J., Morzy, T., Vossen, G. (eds.) ADBIS 2009. LNCS, vol. 5739, pp. 253–269. Springer, Heidelberg (2009)
13. Tucker, P.A., Maier, D., Sheard, T., Fegaras, L.: Exploiting Punctuation Semantics in Continuous Data Streams. IEEE Trans. on Knowl. and Data Eng. 15(3), 555–568 (2003)
14. Bioernstad, B., Pautasso, C., Alonso, G.: Control the flow: How to safely compose streaming services into business processes. In: IEEE SCC, pp. 206–213 (2006)
15. Maier, A., Mitschang, B., Leymann, F., Wolfson, D.: On combining business process integration and etl technologies. In: BTW (2005)
16. Vrhovnik, M., Schwarz, H., Suhre, O., Mitschang, B., Markl, V., Maier, A., Kraft, T.: An approach to optimize data processing in business processes. In: VLDB, pp. 615–626 (2007)