

# StreamMine

**Christof Fetzer, Andrey Brito, Robert Fach, Zbigniew Jerzak<sup>i</sup>**

*Systems Engineering Group, Dresden University of Technology, Germany*

## **KEYWORDS**

Event Stream Processing, publish/subscribe, multi-core, parallelization, speculative processing, Software Transactional Memory, load balancing.

## **ABSTRACT**

StreamMine is a novel distributed event processing system that we are currently developing. It is architected for running on large clusters and uses content-based publish/subscribe middleware for the communication between event processing components. Event processing components need to enforce application-specific ordering constraints, e.g., all events need to be processed in order of some given time stamp. To harness the power of modern multi-core computers, we support the speculative execution of events. Ordering conflicts are detected dynamically and rolled back using a Software Transactional Memory.

## **INTRODUCTION**

The goal of StreamMine project<sup>ii</sup> (<http://streammine.inf.tu-dresden.de/>) is to develop a middleware that supports scalable, near real-time processing of streaming data. As the number of events and data sources increases exponentially, the processing power needed to cope with that information has to be scaled accordingly. For example, for real-time detection of call fraud in a telephone system, one would expect to process from 10,000s to 100,000s events per second. Each event carries a few hundreds of bytes of data and one needs to access several kilobytes of stored data to process an event. Hence, one needs to be able to spread the computation across multiple cores and multiple computers to keep up with this event rate.

The natural choice for scaling up such Event Stream Processing (ESP) applications (Babcock et al., 2002) is to distribute their workload (Cherniak et al., 2003). Such distribution can be performed locally – using multiple processors and processing cores available on a single machine (Welsh et al., 2001). It can also be performed in a distributed environment by using multiple machines connected by a network. Such machines can be either connected by a Local Area Network, forming a cluster (Sterling et al., 1995), or by a Wide Area Network, forming a cloud (Hwang et al., 2007). We architect StreamMine for distributed systems consisting of many-core CPUs. The major contribution of StreamMine is the ability to automatically distribute ESP applications across multiple cores and machines. A StreamMine user is presented with a simple interface that allows for an automatic distribution and parallelization of ESP applications.

The parallelization and distribution are transparent to the user and are performed automatically by StreamMine.

An ESP application consists of a set of connected filter components. In order to harness the processing power of multiple cores per machine, StreamMine can automatically parallelize filters. If the operations to be performed only depend on the input events, i.e., they are stateless, one can trivially parallelize the processing by creating multiple instances of the filter component. However, if there are dependencies between the events, such a simple replication is not a valid approach: processing events out of order can result in incorrect state and incorrect outputs. In other words, StreamMine needs to – at least logically – process events in a given order. Parallelization is a non-trivial problem that can result in significant development costs and runtime overheads. In order to facilitate event processing in multi-core environments, we propose and investigate the speculative execution of events. Speculative execution allows us to process events in parallel that should normally be processed sequentially, even when they are received out-of-order. We use an underlying Software Transactional Memory (STM) (Herlihy & Moss, 1993; Shavit & Touitou, 1995) infrastructure to optimistically process the events in the context of transactions.

The distribution of components across multiple machines poses another set of challenges. The machines used by StreamMine are commodity PCs connected by an IP network. The combination of commodity components and unreliable communication links can result in messages being arbitrarily delayed or dropped. Moreover, there is a very high probability that one or more PCs will crash during an execution. For example, Google's MapReduce jobs, which run on a cutting edge computing infrastructure, suffered in March 2006 on average about 6 worker deaths per job. These jobs were using on average 268 machines and the average completion time was 874 seconds (Dean, 2006).

For ESP applications, we need to expect that the type of processing of the events and hence, the dataflow of the events, will change during the lifetime of a system. In other words, we need to be able to support incremental changes to the system. Besides supporting a simple way of composing event processing applications, we need to support the merging of multiple event processing applications to reduce the computing resources that are required.

Taking above factors into consideration when choosing the communication infrastructure for StreamMine, we have decided to use a Content-Based Publish/Subscribe (Eugster et al., 2003) system as our communication middleware. Content-based publish/subscribe (pub/sub) systems provide a decoupled and flexible communication infrastructure, which allows for easy composition of N-to-M communication patterns. Moreover, thanks to its decoupling properties, it naturally supports dynamic systems, and unlike point-to-point communication schemes, it allows for easy system reconfiguration and merging of computations. In order to cope with potentially large amount of events and components in the system, we propose a novel approach towards construction of the content-based publish/subscribe systems. We show how one can use Bloom filters (Bloom, 1970) to abstract away the content of messages in order to reduce the latency and increase the throughput of events in the system. We also demonstrate an approach to load balancing in publish/subscribe systems. Load balancing

allows us to evenly distribute the load between the components without violating the decoupling properties of the publish/subscribe service.

## RELATED WORK

Very large input data sets and the need to finish processing in a reasonable amount of time were the main driving factors behind batch processing systems such as MapReduce (Dean & Ghemawat, 2008). MapReduce is a programming model for processing large data sets. In the MapReduce model, a user specifies two functions: a *map* function and a *reduce* function. Data is first processed using a *map* function to generate intermediate key-value pairs. Subsequently, the *reduce* function merges all intermediate values associated with the same intermediate key. MapReduce implementation automatically parallelizes and executes user-supplied *map* and *reduce* functions on a cluster of machines.

However, in many application domains, the data source is continuous. A continuous data source implies that the size of the input data is unbounded. Such input data is constantly generated and is usually related to real-time, real-life events. An example of such input data might be the list of outgoing and incoming calls from all mobile phones operated by a certain telephone company. Another example might be the log of transactions performed with credit cards issued by a given bank. In order to be able to work with such data, e.g., to detect credit card fraud or calls made with stolen/hijacked phones, a continuous processing engine for very large amounts of real-time data is required. The approach towards distributed processing of high volume data proposed by the MapReduce system suffers from the fact that it cannot cope with such situations. The main reason being that MapReduce is a staged architecture and, as a consequence, all input has to be processed by the first stage (creating intermediate key-value pairs) before it can be processed by the subsequent, reduce stage. The staged processing introduces unnecessary latencies and such latencies cannot be accepted when we need to guarantee very tight real-time bounds. This assumption is strengthened by the fact that in many applications, delays in the detection of certain conditions can result in monetary loss, e.g., a stolen credit card being used to make purchases or expensive calls made with a stolen cell phone.

The continuous data sources described above are a consequence of omnipresence of sensors, computers and networks and has motivated the establishment of Event Stream Processing (Babcock et al., 2002) as an active field of research. ESP applications target cases such as the telephone or credit card fraud detection described above and many others in fields such as sensor networks or system monitoring. There are many known ESP systems, for example, Borealis (Abadi et al., 2005), StreamFlex (Spring et al., 2007) and GSDM (Koparanova & Risch, 2004). The main difference between these and StreamMine is the way scalability is addressed. The scalability issue may be divided into two sub-problems, one refers to how individual components may be made scalable and the other refers to how to make the communication between components scalable.

Regarding component scalability, Borealis and StreamFlex assume components are normally stateless and, as a consequence, can be parallelized by the creation of replicas for the overloaded components. If components are stateful, Borealis uses load shedding to reduce their load. Nevertheless, if events cannot be dropped randomly, user assistance is

required to decide which events to discard. In GSDM, costly stateful processors are parallelized following a partition-compute-combine approach. This approach, however, can be applied in a very limited set of cases in which two conditions are satisfied: (1) the semantics of the component allows the partition of the input stream and the recombination of the result stream; and, (2) the computational costs of the partition and recombination phases are small compared to the compute phase (otherwise the achievable parallelization would be very limited, according to Amdahl's Law). StreamMine uses an underlying Transactional Memory (TM) (Herlihy & Moss, 1993) as a mechanism for speculation (Brito et al., 2008). This speculation mechanism reduces end-to-end latency by processing events out of their normal order and increases throughput by allowing components to process events in parallel. In addition, these improvements are achieved without user assistance and without waiving sequential semantics. Sequential semantics guarantees that an execution is indistinguishable from a sequential execution, which is important in applications where repeatability and determinism are desired. In addition, it eases the development of the components by not needing the application developers to account for concurrency intricacies.

To achieve communication scalability, StreamMine uses a novel publish/subscribe system that uses Bloom filters to speed up message processing and asynchronous I/O to maximize throughput - see Section "Bloom Filter-Based Routing". The idea behind the publish/subscribe systems has been first framed by the Information Bus system (Oki et al., 1993). Information Bus (and thus pub/sub paradigm) has been motivated by the need to provide continuous operation, dynamic system evolution and interoperability between the dynamic components of distributed systems. The Information Bus proposed a so called topic-based publish/subscribe system, similar in concept to the generative communication model of Linda (Carriero & Gelernter, 1989). The first content-based pub/sub system was proposed in (Rosenblum & Wolf 1997). Since then a number of content-based pub/sub systems has been developed and made available for the research community (Carzaniga et al., 2001; Fidler et al., 2005; Tarkoma, 2008).

One of the main bottlenecks of the publish/subscribe systems has been the speed of the content-based information routing. One of the first approaches towards providing a fast and scalable infrastructure for content-based routing has been presented in (Aguilera et al., 1999), where authors propose to search through the predicates of all subscriptions comparing them with the content of the incoming messages. The presented algorithm's complexity is however linear with the number of subscriptions stored in the brokers. Another approach has been proposed by (Carzaniga & Wolf, 2003). Authors use a version of a counting algorithm, which performs well in case of event forwarding, simultaneously exposing high overhead in case of subscription updates. Orthogonally, (Aekaterinidis & Triantafillou, 2007) proposed the use of Bloom filters to aggregate, filter, and match information in subscriptions, thus creating per broker subscription summaries compacting subscription information. However, the proposed approach requires a known, bounded publication and subscription language, thus limiting the expressiveness of the content-based pub/sub system. The authors of (Cao & Singh, 2005) propose to combine content-based addressing with a multicast based approach. Presented solutions require however that every broker has to be aware of all other brokers in the network, thus breaking the required decoupling of pub/sub components. Authors in (Jerzak & Fetzer, 2007) proposed the use of prefix routing, in order to cope with the

matching complexity. The proposed solution requires a consistent layout of the routing structures across all brokers - ensuring such a common layout might be too expensive in the distributed environment.

Another class of systems are the DHT-based pub/sub solutions (Pietzuch, 2004; Tryfonopoulos et al., 2007). Although they expose superior matching times, their design is based on topics. Moreover, a failure of a single node might seize the delivery of messages for all topics a given broker manages. A similar approach has been presented in (Cutting et al., 2008) where instead of explicit topics authors use tags, which designate the rendezvous nodes for the publications in the peer-to-peer system.

## ARCHITECTURE

A typical event processing application is structured as a cascade as depicted in Figure 1. Producers (e.g., sensors) generate events that are processed by a sequence of components until a consumer is reached (e.g., a monitor dashboard, an actuator back in the observed system). During this process, the event streams may be joined or split. Events can also cause the generation of newer events, have some offline information added to them (e.g., from a database) or be filtered out of the stream.

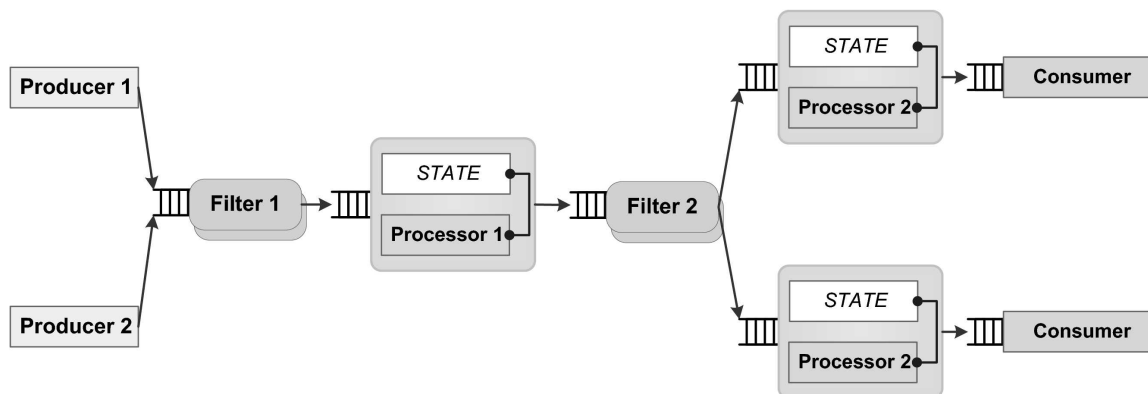


Figure 1. Logical view of a typical event stream-processing network.

In StreamMine, although the application is logically represented by a dataflow, the underlying physical connections resemble a cloud. Figure 2 depicts a possible physical materialization of the system from Figure 1. The elements inside the cloud are the brokers from the publish/subscribe (pub/sub) middleware, which are responsible for efficiently routing messages to and from each component.

The main advantage of using a pub/sub infrastructure is the loose coupling between components. In StreamMine, components can be inserted and removed at runtime by other components. This feature is essential for building systems that can scale up and down by orders of magnitude. For example, a Filter component that executes a costly operation may need to be replicated such that its workload will be split between itself and the new replica. With pub/sub this load balancing is possible by simply initiating a new replica (either at the local site or in a remote one) and creating a subscription for the new replica.

Components can also be replicated for fault-tolerance purposes. In this case, the replicas will have the same subscription as the original so that they process the same subset of events. A resulting requirement is that events have ids that allow duplicated events to be discarded. By using these ids, when more than one replica of a fault-tolerant component is operational, the replicated results can be discarded when they reach the next component.

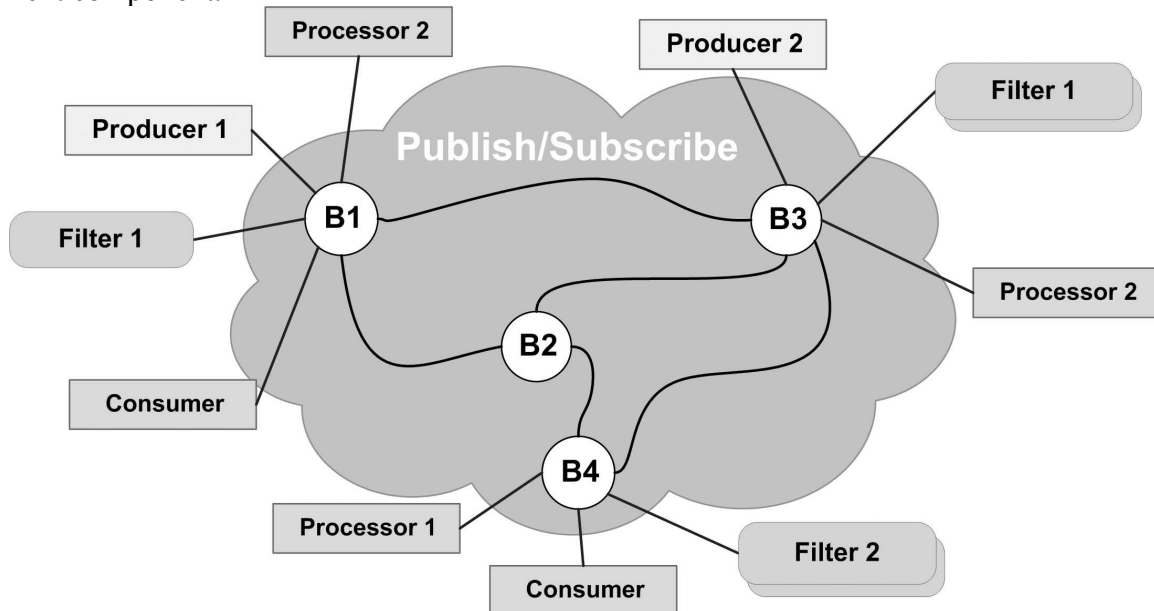


Figure 2. Physical deployment of the logical network of Figure 1.

In event stream applications, a common type of stateful component is one that learns from the stream. Components learn by building (limited-size) sketches of the event stream or by fitting it to a model. This is an example of a case where the order that events are processed may affect the result. There are three main reasons why total ordering among events is necessary: (1) consistency, if such a stateful component is replicated for fault tolerance, all the components must process the events in the same order; (2) determinism, it is frequently desirable that the system can be replayed, i.e., the same sequence of events result in the same outputs; (3) correctness, for example, if the component computes a trend in the data, then, processing two events in the inverted order (e.g., because they were previously processed by different replicas of a preceding component and took different paths or different processing times, which resulted in an order inversion) leads to an incorrect result. StreamMine supports the use of unique timestamps for ordering events. These unique timestamps are typically given by the application but these can also be generated by StreamMine if needed.

## BLOOM FILTER-BASED ROUTING

### Overview

The ultimate goal of using a pub/sub service in the context of StreamMine is to provide a (1) fast, (2) flexible, and (3) decoupled infrastructure for information exchange between

components. The pub/sub communication model is well suited for the highly dynamic ESP processing systems. It is asynchronous: neither publishers (data producers) nor subscribers (data consumers) are blocked when producing or receiving data. Both subscribers and publishers are anonymous to each other and the system as a whole. There is also no requirement on the communication between any two components to take place simultaneously. In content-based publish/subscribe systems subscribers express their interest in a given data using subscriptions. Publishers use events to disseminate the information they have produced.

StreamMine follows a common convention (Carzaniga et al., 2001; Pietzuch, 2004; Zhao et al., 2004; Fidler et al., 2005; Tarkoma, 2008; Jerzak & Fetzer 2008), where subscription  $s$  is a conjunction of predicates. A predicate  $p$  is a function which evaluates to either true or false for a given argument. Predicates consist of attribute names and attribute constraints. An attribute constraint, in turn, consists of an operator and an attribute value. An event  $e$  is a set of attribute name and attribute value pairs. Attribute name and value pairs forming an event are arguments for the predicate functions. A sample subscription can be represented as:  $s \equiv \{x > 5, y > 3\}$ . This sample subscription  $s$  consists of conjunction of two predicates  $x > 5$  and  $y > 3$ . The attribute name of the first predicate is  $x$ . The attribute constraint of the first predicate equals  $>5$ , where  $5$  is the attribute value and  $>$  is an operator. A sample event can be depicted as:  $e \equiv \{x 7, y 5, z 3\}$ . An important concept regarding the subscriptions is the binary coverage relation. We can say that subscription  $s1$  covers subscription  $s2$  if subscription  $s1$  matches the superset of events matched by subscription  $s2$ . We might also say that  $s1$  is more general than  $s2$ .

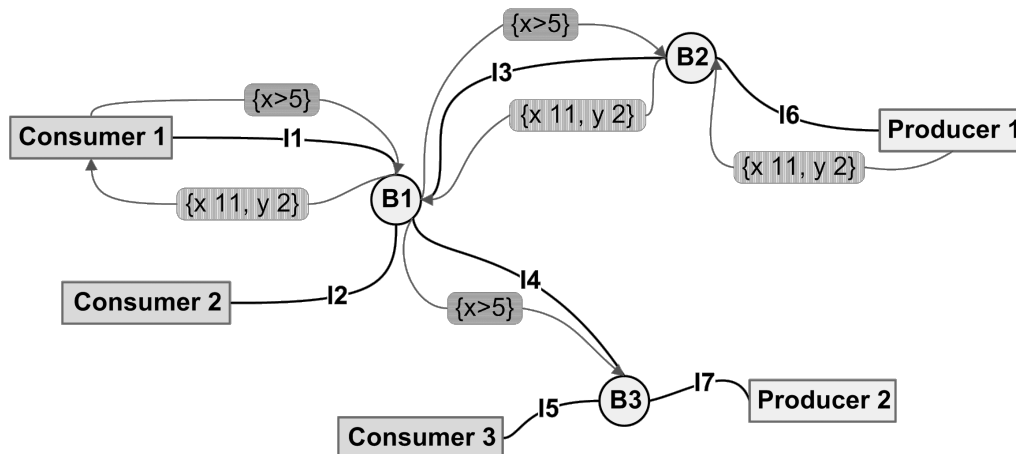


Figure 3. Routing of subscriptions and forwarding of publications.

Subscriptions and events allow for dynamic information routing as the explicit source and destination addresses usually put on events are made obsolete by the content of the events. Hence, the potential failures of publishers or subscribers are naturally masked by the pub/sub communication middleware. A subscription  $s$  matches event  $e$  if every predicate contained in the subscription evaluates to true over the content of the whole event.

An example of content-based information exchange in a publish/subscribe system has been presented in Figure 3. *Consumer 1* issues the subscription  $\{x > 5\}$  which is

distributed into the network of brokers  $B1$ ,  $B2$  and  $B3$ . Subsequently, Producer 1 issues the event  $\{x\ 11, y\ 2\}$  which is delivered by the broker network to the interested consumer. Please note that the subscription  $\{x > 5\}$  has to be delivered to all brokers in the network so that if *Producer 1* changes its location from  $B1$  to  $B2$  or  $B3$  its events can still be transparently delivered to the *Consumer 1*.

In StreamMine, publishers and subscribers can be placed on different physical machines. Every machine runs a broker process which is responsible for the subscription routing and event forwarding. In the process of subscription routing every broker in the StreamMine network stores a set  $S$  of subscriptions which allows it to forward events to all interested subscribers from all potential publishers on the reverse paths of stored subscriptions.

### **Content Summarization using Bloom Filters**

Current approaches (Carzaniga et al., 2001; Tarkoma & Kangasharju, 2006; Mühl et al., 2006) to content-based routing assume that events on their way from publisher to subscriber are matched based on their content at every broker they pass. If we refer to the example shown in Figure 3, we can observe that the event  $\{x\ 11, y\ 2\}$  needs to be matched against the subscription  $\{x > 5\}$  by both brokers  $B1$  and  $B2$ . Such an approach has two main issues: (1) the evaluation of predicate functions over events' content is slow, and (2) the slow evaluation has to be performed by every broker an event is forwarded to. Thus, the more brokers an event passes on its way, the higher the delay and the processing overhead.

We propose a new routing algorithms and accompanying data structures for use in StreamMine publish/subscribe communication middleware. The new approach allows us to route events evaluating the subscriptions' predicates over their content only once. This evaluation can take place at the first broker in the network or already at the publisher, so that no content-based matching needs to be performed at any of the brokers between the publisher and the subscriber.

We achieve this goal by using content summaries. Content summaries are compact content representation using integer values, thus greatly increasing the speed of event matching. Specifically, we do not use the rendezvous approach (Pietzuch, 2004; Cutting et al., 2008) and thus do not rely on any single broker in the network to be the matching point for a given class of events.

The routing processes requires the presence of subscriptions. Subscriptions arriving at the broker are stored for the purpose of matching against incoming events. An event arriving at the edge broker has to be matched based on its content with the stored subscriptions. The result of the content-based match is encoded in a Bloom filter subsequently attached to the event. Downstream brokers encountered by the event evaluate only the attached Bloom filter. Specifically, they do not consider the content of the event.

Let us once again consider the example shown in Figure 3. In case of event  $\{x\ 11, y\ 2\}$  it would be first forwarded to broker  $B2$ . Broker  $B2$  uses the content of the event in combination with the set of subscriptions it stores to compute a Bloom filter summarizing the content of the event. This Bloom filter is subsequently attached to the event by the broker  $B2$ . In the next step broker  $B2$  performs the matching process based only on the contents of the Bloom filter attached to the event. The result of the matching

process is a set of brokers (in case of Figure 3 it is the broker *BI*) to forward the event  $\{x\ 11, y\ 2\}$  to. Broker *BI* upon reception of the event  $\{x\ 11, y\ 2\}$  with attached Bloom filter only needs to perform the matching process based on the contents of the attached Bloom filter. Therefore, broker *BI* does not need to consider the content of the event.

For the above scenario to work we have developed a novel data structures and algorithms for storing subscriptions at the publish/subscribe brokers and for matching of the incoming events. In the following sections we will describe in detail the data structures used to store the subscriptions in the publish/subscribe brokers, as well as the event matching algorithms.

### Subscription Content Summarization

StreamMine brokers store subscriptions in the `sbsposet` data structure. The `sbsposet` stores subscriptions' predicates by abstracting away the conjunctive form of subscriptions, which allows for a more efficient storage structure - see Figure 4. The `sbsposet` stores predicates grouped by their attribute name. In the Figure 4 we can observe two attribute names: *x* and *y*. Every attribute name points to a partially ordered set (poset). Each poset begins with a virtual root node (*null*) and stores attribute constraints sharing the same attribute name. The structure of the attribute constraints reflects the partial order resulting from the covering relation between them. Every attribute constraint has a Bloom filter associated with it. Bloom filters are created when attribute constraints are inserted into the `sbsposet`. On Figure 4 Bloom filters of attribute constraints are represented by the indices of the bits set to one, enclosed in the curly braces.

The creation of Bloom filters for attribute constraints is performed by using the whole predicate containing the given attribute constraint as the input to the Bloom filter. Therefore, for the attribute constraint  $>5$  the Bloom filter is constructed by hashing the  $x>5$  string. The example subscriptions and corresponding Bloom filters are shown in Table 1. Throughout this chapter, unless otherwise stated, we use Bloom filters with  $k=2$  hash functions with the width of  $m=2^{14}$  bits.

The Bloom filters of attribute constraints' in the `sbsposet` maintain the coverage relation between the attribute constraints belonging to a respective attribute name. Every attribute constraint covered by another attribute constraint reflects this fact in its own Bloom filter by performing an OR operation with the Bloom filter of the covering attribute constraint. A single Bloom filter reflects the whole chain of covering attribute constraints.

Subscription	Source	Bloom filter
$\{x > 5\}$	f1@dot.com	6066, 8581
$\{x \geq 0.7\}$	f2@dot.com	2787, 12518
$\{x > 15\}$	f3@dot.com	6441, 10582
$\{y < 5\}$	f4@dot.com	5037, 8516
$\{y = 0\}$	f5@dot.com	5001, 8507
$\{x > 15, y > 0\}$	f6@dot.com	5102, 6441, 8636, 10582

Table 1 - The set of subscriptions used in Figure 4, Figure 5 and Figure 6

The `sbsposet` presented on Figure 4 contains subscriptions presented in Table 1. We can observe that every covered attribute constraint (e.g.,  $>5$ ) contains the bits from the covering attribute constraint ( $\geq 0.7$ ) Bloom filter. An interesting property of the `sbsposet` is that by removing the conjunction between predicates of a single subscription it can reduce the number of stored predicates. A simple example are the subscriptions  $\{x > 15, y > 0\}$  and  $\{x > 15\}$  from the Table 1. The `sbsposet` needs only 2 predicates to store both subscriptions, while a naive approach would require the storage of all three of them.

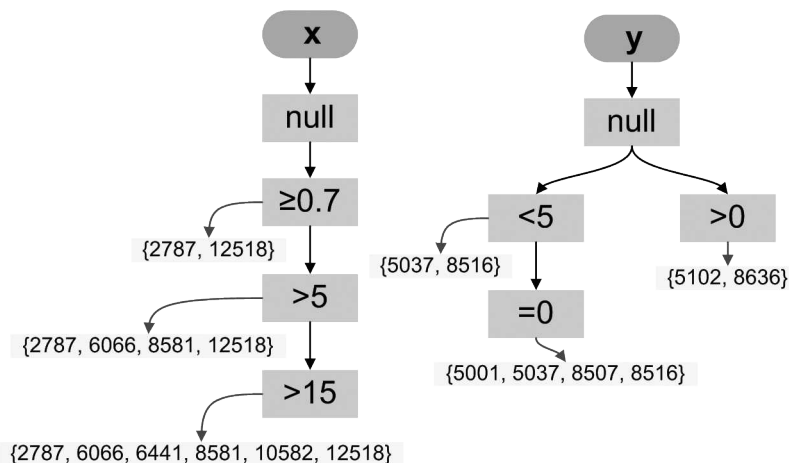


Figure 4. The `sbsposet` storing subscriptions from Table 1.

The addition of a new subscription into the `sbsposet` is performed by decomposing it into single predicates and inserting the predicates on a one-by-one basis into the appropriate branches. The removal of subscriptions from the `sbsposet` using a counting Bloom filter (Fan et al, 2000) is also straightforward, however for the sake of the presentation brevity we omit the details of the implementation.

The matching of events with the `sbsposet` must ensure that every event encodes in its Bloom filter all predicates matching its content. Therefore, for every predicate in the the `sbsposet` matching the event's content, the corresponding predicate's Bloom filter is added (using an OR operation) to the (initially empty) Bloom filter of the event. By storing the Bloom filters of all covering predicates in every predicate in the `sbsposet` we can optimize the process of addition by performing it only once per matching predicate and simultaneously encoding a possibly large number of covering predicates.

As an example let us consider an event  $e \equiv \{x 7, y 8\}$  and the `sbsposet` presented in Figure 4. Initially the Bloom filter of the event is empty. The matching of the event starts with the  $x 7$  attribute name and value. This attribute name and value selects the  $>5$  attribute constraint from the  $x$  poset. The Bloom filter of the selected attribute constraint is Ored with the (so far empty) Bloom filter of the event, with result being stored in the Bloom filter of the event. Now the Bloom filter of the event reads  $\{2787, 6066, 8581, 12518\}$ . The matching of the  $y 8$  attribute name and value pair is performed in a similar manner. The  $y 8$  attribute name and value pair selects  $>0$  attribute

constraint from the  $y$  poset. Hence, the Bloom filter of the event  $e$  finally contains:  $\{2787, 5102, 6066, 8581, 8636, 12518\}$ .

## Event Content Summarization

The `sbsposet` stores the content of the subscriptions without regarding the conjunctions between the predicates of a single subscription. The loss of this information could lead to a potentially large number of false positives, i.e., events delivered to subscribers which did not subscribe to them. Hence, there is a need for a data structure which would represent the conjunction between the predicates of subscriptions. In this section we introduce such data structure: the `sbstree`. The main task of the `sbstree` is to represent the disjunction of conjunctions of predicate values. Therefore, in contrast to the `sbsposet`, the `sbstree` stores subscriptions in their conjunctive form. Specifically, the `sbstree` does not store any predicates, instead, it works exclusively with the Bloom filters representing subscriptions and events.

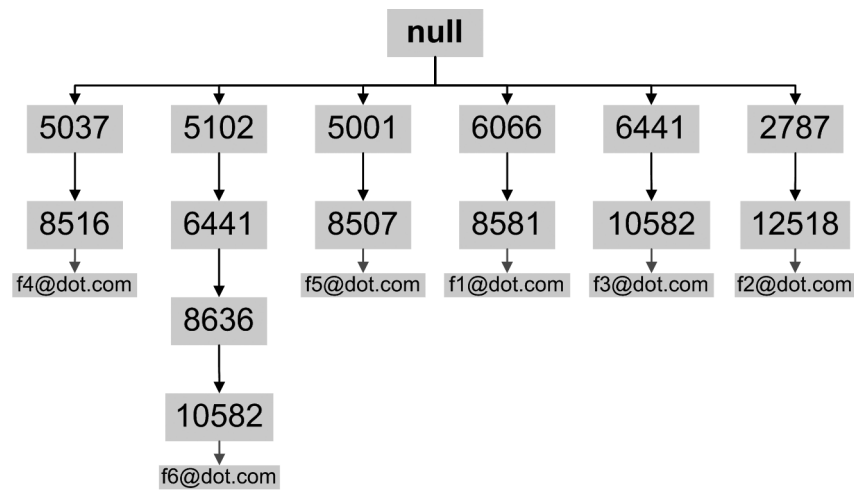


Figure 5. The `sbstree` storing subscriptions from Table 1.

The disjunction of the predicates of a single subscription can be expressed with a single Bloom filter using an *OR* operation between the Bloom filters of single predicates. However, this approach cannot be used to represent a conjunction of subscription predicates. Therefore, we use the `sbstree` structure to cope with that issue - see Figure 5. The path from the root to the leaf of the `sbstree` forms a conjunction of the subscription predicates, or more precisely, the conjunction between the bits set in the subscriptions' Bloom filters.

A Bloom filter of a subscription is an *OR* between the Bloom filters of the subscriptions' predicates. For a given subscription the bits set in its Bloom filter form a path rooted at the virtual `sbstree` root node - *null*. The leaf of the path represents an interface on which the given subscription arrived. Hence, a path leading to the source interface forms a conjunction of all predicates of a given subscription.

The event matching process starts at the virtual root of the `sbstree`. For every integer representing a bit set in the Bloom filter of an event, a comparison with children of the root node is performed. In case of a match, the given path is followed until either:

(1) the end of the path (indicated by the source interface) is reached or (2) a given node in the path does not have a corresponding value in the event's Bloom filter.

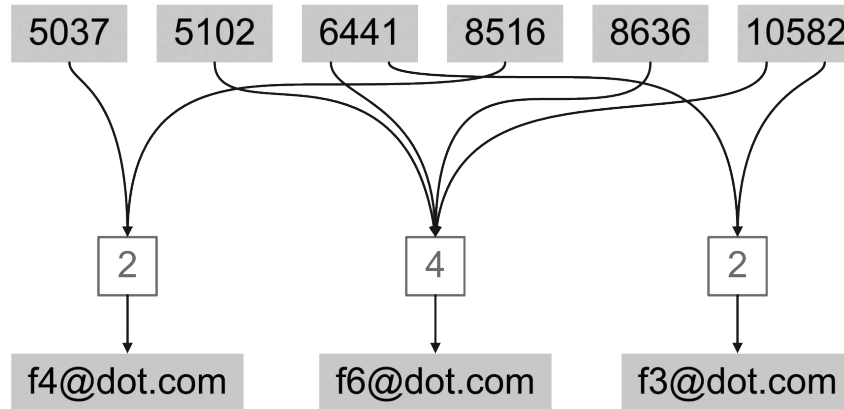


Figure 6. The counting algorithm variant of the sbstree from Figure 5 storing three subscriptions from Table 1.

The above algorithm suffers however from the fact that for every bit set in the event's Bloom filter a path in the sbstree needs to be selected and possibly followed. From the sbstree traversal algorithm we can conclude that following a given path does not guarantee the reaching of the source interface at the end of it. Additionally, the complexity of the sbstree traversal grows exponentially with the number of bits set in the event - as more and more paths need to be followed.

Therefore, we have developed an improved version of the sbstree, based on a counting algorithm (Pereira et al., 2000; Fabret et al., 2001). The new version of the algorithm stores all bits set in all subscriptions' Bloom filters in a linear fashion - top row in Figure 6. All bits from a Bloom filter forming a subscription are assigned a counter. The value of the counter is equal to the number of bits set in the subscription's Bloom filter - middle row. For the subscription which arrived at interface *f6@dot.com* this value equals 4, as it has two predicates, each composed of a Bloom filter having 2 bits set. The counter itself represents a set of bits which need to be present in the event in order to trigger the subscription it is connected to.

Upon arrival of an event its Bloom filter is parsed for the values of set bits. Every bit set in the event's Bloom filter triggers the corresponding bit in the counting sbstree. Triggering a bit corresponds to the decreasing of the counter which is connected to it. Whenever a counter reaches 0 the corresponding subscription matches the given event. Specifically, event can be forwarded to the interface on which the matching subscription arrived - bottom row in Figure 6.

The above algorithm, in contrast to the original sbstree variant, is linear with the number of bits set in the event. In Section "Measurements" we present the evaluation of this approach and contrast it with the original sbstree.

## STM-BASED PARALLELIZATION

## Overview

As discussed earlier, stateful components cannot be parallelized by simple replication. As a consequence, stateful components are often the bottlenecks of event processing applications. A component is classified as stateful if the output for the input event  $e_t$ , where  $t$  is the timestamp of the event, may depend not only on the event itself, but also on the current state of the component, which is function of previous events. Having the application programmer design a component that is already parallel can increase too much the costs due to the decrease in productivity. For example, when developing a parallel component for a multiprocessor machine, a programmer can use coarse locks to control the concurrent access to the shared state, but that will result in small, if any, gains. On the other hand, if the programmer uses fine-grained locking or lock-free algorithms, the development gets too complex and too error-prone.

In some cases, stateful components that operate over single events can be transformed in stateless components that operate over a small set of events. In order for this to be possible, there must be no dependence between two sets of events, for example, computing the moving average of the value attribute of a stream of events using jumping windows. However, the same approach is not valid for sliding windows. In some other cases, known as partition-compute-combine, the stateful component can be replicated as long as the events processed by each replica are chosen by a partition algorithm and their results are merged by a combine algorithm (Koparanova & Risch, 2004). Unfortunately, there are no general algorithms for the partition and combine phases, what is required is the application programmer to provide these two components in addition to the original stateful component.

Thus, if the main goal is to provide an infrastructure that allows application programmers to build scalable applications in a productive way, we should not require that they understand all the intricacies of parallel and distributed computing, and then automatic parallelization is the only solution left. Parallelizing compilers have been investigated for a long time, but because the decisions are made in compilation-time, when much less information is available, they are pessimistic and, thus, explore less parallelism than indeed available.

## Optimistic parallelization of stateful components

The approach used in StreamMine is to use speculative execution to process events in parallel. The main idea is to use components that were not designed to be parallel and, thus, are easier to design. When two events are executed concurrently, they execute in sandboxes that isolate one from the other. During the processing, if they do not interfere with each other, they are said to be non-conflicting. After finishing processing, results of non-conflicting executions can be merged into memory. On the other hand, if they conflicted, the execution with less priority (e.g., the execution of the later event) is aborted and scheduled to be executed after the conflicting one. The main requirement is that all events are executed in a sandboxed environment and their modifications to the system state are made visible only in the order of their timestamps. As a consequence, the history of component states is the same as in a sequential execution.

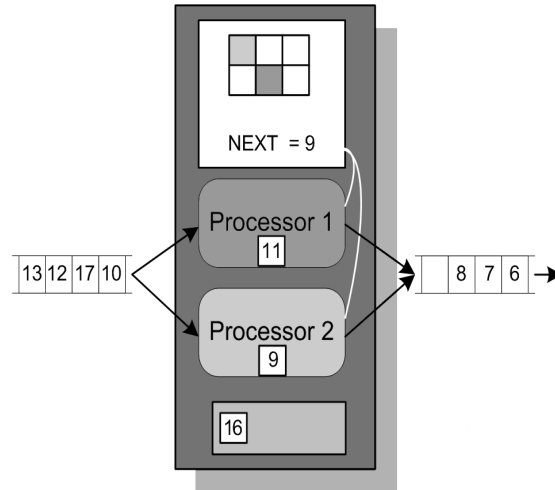


Figure 7. Structure of a speculative component.

This speculative execution is enabled by an underlying Software Transactional Memory (STM). The STM intercepts the memory accesses (reads, writes, memory allocations and releases) and check against interferences. Consider for example the stateful component *Processor1* in Figure 1, the internal structure of this component is shown in Figure 7. The state of the component is divided into 6 parts (e.g., 6 fields or 6 objects) and two events  $e_9$  and  $e_{11}$  are being processed concurrently. If (a)  $e_9$  accesses one part of the state and  $e_{11}$  accesses another, as shown in Figure 7, there is no interference and both computations are successful. Otherwise, if (b)  $e_9$  writes to a memory position that was already read by  $e_{11}$ , there was interference because  $e_{11}$  should have considered the value written by  $e_9$  and thus,  $e_{11}$  must abort and re-execute. However, if (c)  $e_9$  reads from a position that was modified by  $e_{11}$  it can still retrieve the original values as modifications from  $e_{11}$  are still not incorporated to the component state yet (i.e., they are still confined in the sandbox). Further, (d) there is also no interference if both write to the same position (and none of them read), because the written values will only be incorporated to the component state at the timestamp order and, thus, the write from the later event will overwrite the write from the earlier one as expected. Finally, (e) there is no interference if both read from the component's state.

In addition, the illusion of a sequential execution is provided by having a counter that indicates the timestamp whose effects should be made visible next (shown as  $NEXT=9$  in Figure 7). Events that finished processing, but that cannot be made visible yet are stored in a waiting set, as is the case of  $e_{16}$  in Figure 7.

The benefits of the speculation affect both latency and throughput. First, consider the case that a stateful component (such as component *Processor1* in Figure 1) receives messages from replicated components (such as the replicated filters in the same figure). If the semantics and the workload are such that there is some parallelism available (i.e., there are cases in which two different events do not conflict), then processing these events out of their normal order will decrease their processing latency. Additionally, because more than one event can be processed at the same time, throughput is also improved.

## Limiting speculation

A point that is still open regards how speculative should the system be. Speculatively processing event  $e_{200}$  when the last changes written to the component's state were from event  $e_{10}$  may not be a good decision. After the processing from  $e_{200}$  is finished, its modifications will only be visible after all the events with timestamps in between 10 and 200 are also processed. Thus, if  $e_{200}$  conflicts with any of these missing events its processing will be discarded. To decide how much parallelism the combination component and workload has and how far should the system speculate, we use *conflict predictors* (Brito et al., 2008).

A conflict predictor looks at all events available, in other words, both the events not yet processed and the events processed but also aborted (e.g., because of conflicts) and decides based on the available resources which one should be processed next. The specification of the conflict predictor is a set of rules that groups events into classes and establishes a speculation horizon for each class. This set may combine user-provided rules, rules derived from static analysis and rules generated by runtime analysis. By default, no static analysis or user-provided rules are used, just a simple but useful predictor is available. This default predictor puts all events in the same class and tries to keep the number of aborts low by dynamically changing the speculation horizon. Through this approach, even if it the workload does not exhibit any parallelism, execution performance will still be as good as a sequential execution. However, if there is parallelism, at least part of it will be harnessed. This predictor works by keeping the numbers of aborts close, but not equal, to zero. If there are no aborts, it is an indication that perhaps more speculation can be done. Otherwise, if the number of aborts is too high, too much speculative work is being discarded and this impacts negatively on system performance. Although such predictor is likely to have a suboptimal performance, it allows out-of-the-box sequential components to achieve immediate gain.

## Distributed speculation

Finally, the system gives the user the option to enable that speculative results are propagated to downstream components. If this is the case, speculative components will forward speculative results and will later send updates to these events as they get aborted and executed and when they reach a final version. It is up to the user to mark which components should accept speculative results. For example, if some component is a gateway to a physical system the application programmer may not want that actions are taken based on speculative results. By not marking a component to accept speculative events, the component will ignore all events that are not marked as final. Components that are marked as speculative, however, can handle speculative events very efficiently. Because the underlying STM monitors all the reads and writes from events it processes, the STM can efficiently detect if the update received for a previous speculative version really requires a re-execution. For example, if the final version of an event differs from the previous version only by some attribute value and this attribute was not read by the current execution, then there is no need to abort and re-execute the current event as the results would be the same.

Because of the speculation support for stateful components, events can also be processed out of their normal order even in cases that the component semantics requires ordered processing. As mentioned in previously, if stateful components are replicated for fault tolerance, all the replicas must process the events in the same order and that requires costly atomic broadcast protocols. Speculation becomes then very helpful to hide such costs. On the one hand, if events have timestamps that describe a total order, these events can be simply submitted to component as they arrive and the speculation support will ensure that the component state and, consequently, the result of the computation will obey this total ordering and thus, be the same in all replicas. On the other hand, if the component does not require a total ordering the atomic broadcast protocol may use an optimistic delivery to submit events as speculative to the component, assign the optimistic order as the timestamps and later deliver the non-speculative versions with the correct order. Then, even if events were optimistically delivered in the wrong order, the computations will be checked and consistency guaranteed.

## **LOAD BALANCING**

Two of the main design goals for StreamMine are scalability and near real-time event processing. In this section, we illustrate these two design goals using a real-world application for the StreamMine framework.

Our example application is a Quality of Service monitoring tool for prepaid telephone services. The goal of the tool is to supervise the success rate of reload transactions for prepaid mobile phones - see (Campanile et al., 2008) for more details. Typically, several distributed services participate in the reload transaction process. Such a transaction is considered to be successful if all participating services have executed their participating services with success in a timely fashion. Hence, the basic idea for the Quality of Service monitoring tool is to publish information about the reload process of all involved services and to analyze these events using a Timed Finite State Machine (TFSM). Those state machines have states and transitions for a successful or failed reload of one specific transaction and are automatically created for each new reload transaction. Obviously, there might be Quality of Service requirements that need near real-time processing of events in order to detect the success or failure of a specific ongoing transaction. Another problem arises with the huge amount of transactions that are executed in parallel. As mentioned before, each such transaction will have its own TFSM. This results in unacceptable high load if the state machines were to be executed on a single node. Hence, we need to support the scalability of the monitoring process to an arbitrarily high number of parallel transactions. To ensure real-time processing at very high event frequencies, the load balancing scheme of StreamMine has been introduced.

In StreamMine, we balance the load of specific components on the nodes by replicating the component to another node and by partitioning the event flow toward these components. This could be done by partitioning the event domain over these components using an appropriate subscription for each state machine. However, this approach may result in too large number of subscriptions, is not scalable and can become complicated for some attribute domains, e.g., strings. Therefore, we have introduced a hash-based partitioning scheme as part of the publish/subscribe, which allows balancing the load while limiting the amount of required subscriptions.

Figure 8 presents the logical view on a deployment of a QoS monitoring network. *Source 1* and *Source 2* represent event sources of monitored services and these implicitly act as load balancing proxies. Boxes in the middle are monitoring components each one executed by different nodes hosting a couple of FSMs, in general cluster nodes. As mentioned above, each FSM is responsible for observing the QoS of one transaction. Monitoring results are published by the FSMs and are received by the consumers. Each event containing a new transaction identifier triggers the creation of a corresponding FSM by the monitoring component. The hash-based load balancing is used to partition the event flow between the two event sources and the two monitoring components at nodes  $N_i$  resulting in an approximately even load distribution. In the following, the StreamMine load balancing architecture will be presented.

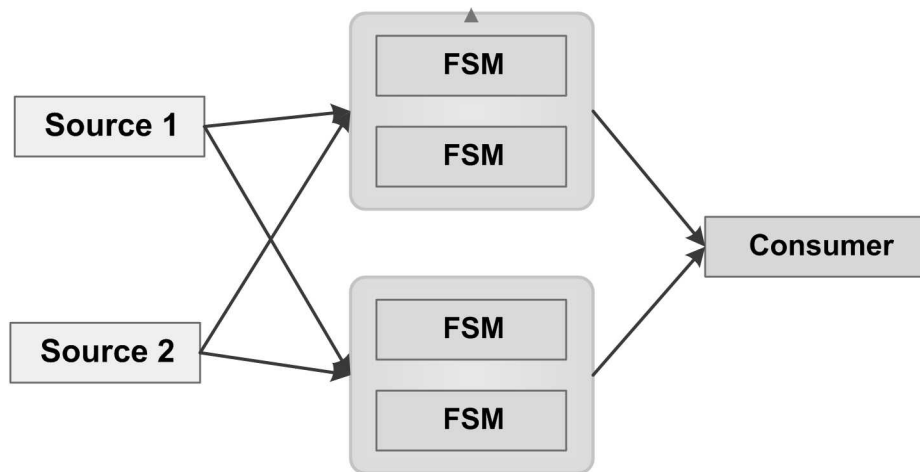


Figure 8. Logical view of a Quality of Service Monitoring Network.

As already mentioned, in StreamMine publish/subscribe is used as the underlying communication abstraction. Let  $e = \{txid, \dots\}$  be an event issued by load balancing proxy and  $s = \{hfn(txid) \bmod N\}$  be a subscription issued by cluster nodes, whereas  $txid$  is the key for the partitioning, here the transaction identifier and  $N$  is the total number of cluster nodes. Our initial approach only supports static load balancing. In future, we will extend StreamMine to support dynamic load balancing too.

## MEASUREMENTS

In order to evaluate our approach for building of the StreamMine processing system we have performed a series of measurements exposing the benefits and performance of the components of the StreamMine platform. Unless otherwise stated all measurements have been conducted using the StreamMine cluster of the Systems Engineering group at the Dresden University of Technology.

### Microbenchmarks

In order to highlight the importance of the counting `sbstree` optimization we have performed a test in which we have varied the average number of bits set per event while maintaining a constant number of subscriptions in the original `sbstree` and counting

`sbstree` - see Figure 9. The average number of bits set per event corresponds to the number of subscriptions matching a given event, divided by the average number of predicates per subscription times the number of bits per predicate.

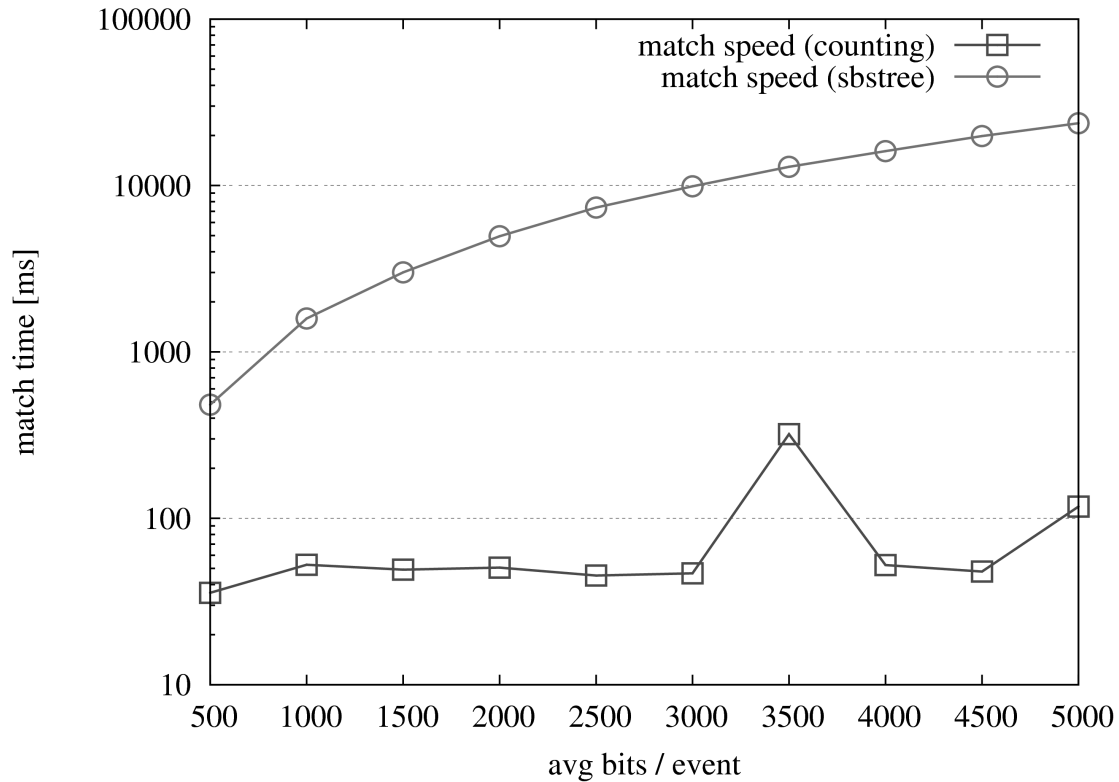


Figure 9. `sbstree` and the counting `sbstree` data structures performance.

We can observe that the matching time in case of the original `sbstree` (indicated as `sbstree` in the Figure 9) increases exponentially with the number of bits set in the event. The optimized variant (indicated as `counting` in the Figure 9), using a counting algorithm is linear with the number of bits set.

In order to compare the behavior of both variants of the `sbstree` with the traditional matching algorithm presented in (Tarkoma & Kangasharju, 2006) we have performed an experiment shown in Figure 10. In this tests we were varying the sizes of the respective matching structures - `forest` for (Tarkoma & Kangasharju, 2006), `sbstree` for the original variant of the `sbstree` and `counting` for the optimized version of the `sbstree` - by inserting increasing number of random subscriptions. For every new size of the routing structure we have matched 1000 random events and plotted the cumulative matching time of 1000 events.

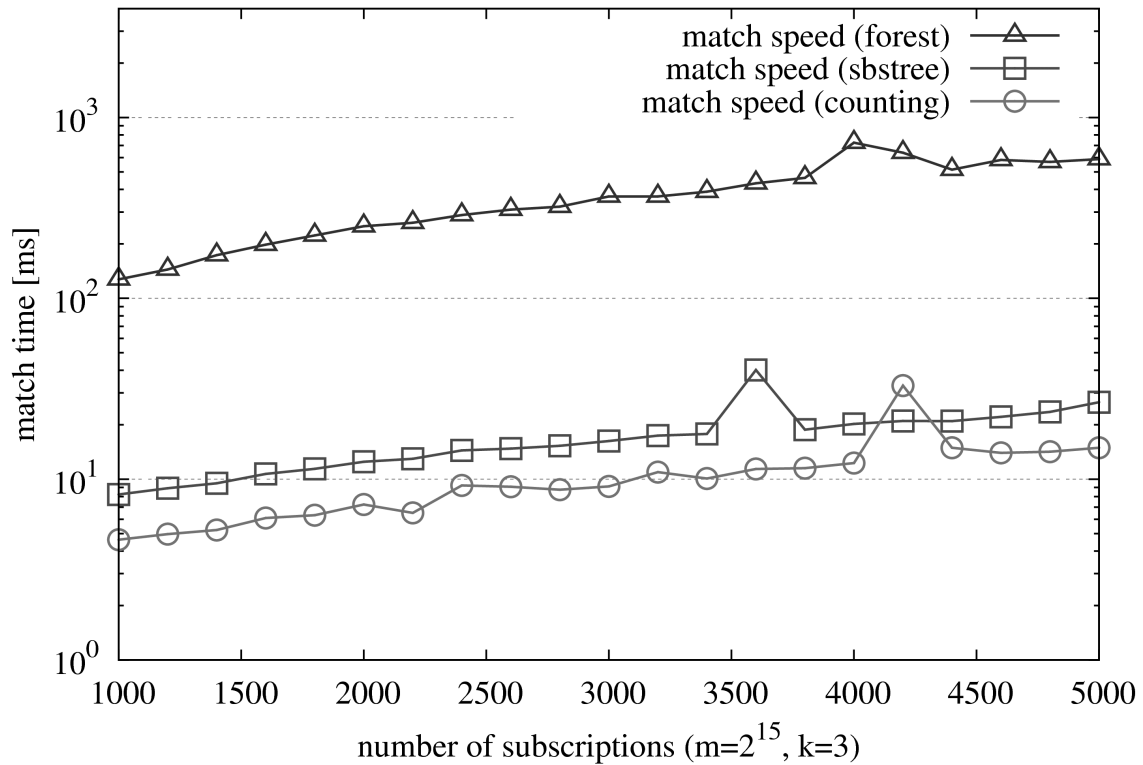


Figure 10. Comparison of matching times of different data structures

The subscriptions and events were created using 20,000 unique attribute names. Attribute names for both events and subscriptions are selected from the Automatically Generated Inflection Database (AGID)<sup>iii</sup>, based on aspell word list, containing 112,505 English words and acronyms. Subscriptions consist of 2 to 5 predicates. The number of predicates is selected using a uniform distribution. The likelihood of subscription coverage and event matching has been set up using the Pareto distribution. The Pareto distribution used in all tests had the parameters  $\alpha=1$  and  $\beta=3$ <sup>iv</sup>.

Figure 10 clearly illustrates the improvement in the matching times when using the content summarization techniques - especially in comparison to the traditional, full content-based approach - `forest`. We can observe that the counting version of the `sbstree` performs better in comparison to the original `sbstree`, due to the simpler data structure and the linear matching time algorithm.

### Latency and throughput

In the following tests we have focused on two aspects on the functioning of the publish/subscribe communication substrate - the (1) latency and (2) throughput as perceived by subscribers in a real, functioning deployment. For the following experiment we have created three networks: 1 broker, 2 brokers and 3 brokers - see Figure 11. In every network we have deployed one subscriber (*s1*) and one publisher (*p1*) connected by an increasing number of brokers between both of them.

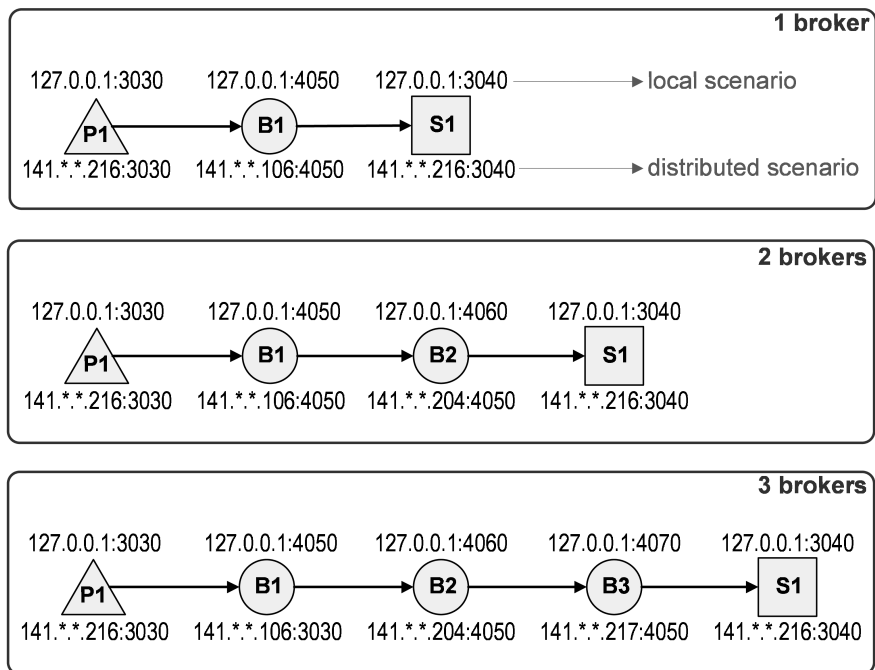


Figure 11. The experiment setup.

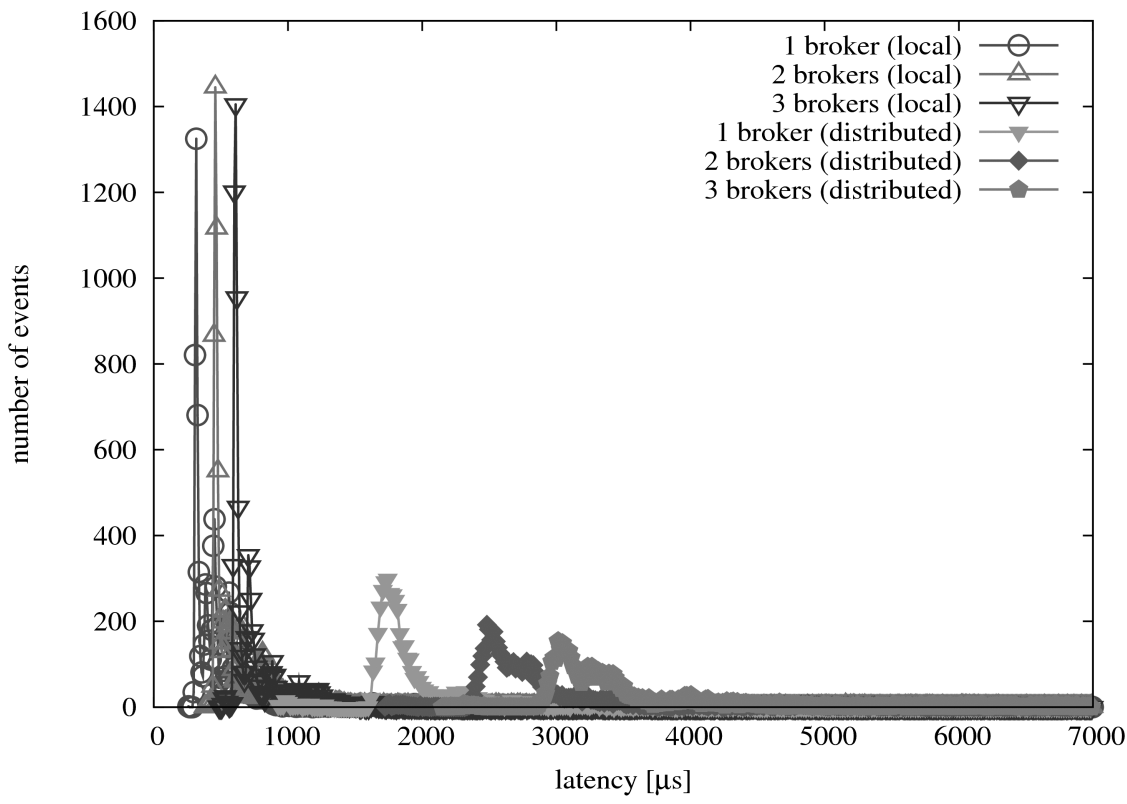


Figure 12. Histogram of the event forwarding latencies.

Every network setup has been executed in two settings - local and distributed. In the local setting we have created the networks one machine and connected their components via a loopback interface. For the distributed scenario every component was placed on a separate node of the StreamMine cluster.

Figure 12 compares the event forwarding latencies perceived by subscribers in different scenarios. We can observe that the increase in latency is proportional to the number of intermediate brokers and in case of the distributed scenario is dominated by the network latency. For the latency measurement we have assumed 200 different subscribers connected to each broker.

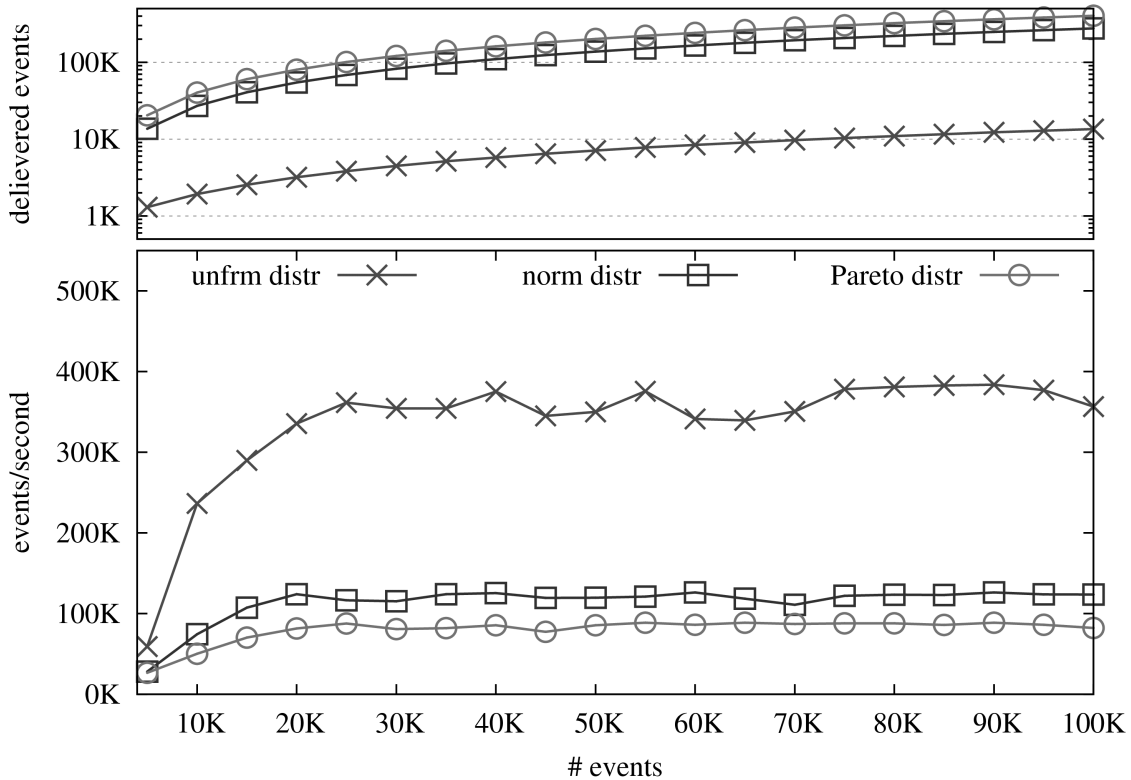


Figure 13. Event throughput.

For the throughput measurements (see Figure 13) we have evaluated the counting `sbstree` routing structure performance without the influence of the networking components. We have connected a subscribers directly to the publish/subscribe broker - which implies direct event delivery using upcalls. The subscriber issued 200 different subscriptions. Subsequently, we have attached one publisher which issued events matching varying number of the subscriptions. Figure 13 shows the throughput of the whole setup in terms of events per second received by the subscriber. The three different distributions: uniform (`unfrm distr`), normal (`norm distr`) and Pareto (`Pareto distr`) determine the overlap between the events and subscriptions, i.e., they determine the likelihood of an event being matched by a subscription and subsequently delivered to the subscriber. In the upper part of the graph the total amount of events delivered to the subscriber has been plotted. We can observe that the throughput is proportional to the

number of delivered events, as more and more upcalls need to be performed by the broker. The above test has been performed on a commodity PC. We can therefore conclude that the raw throughput of the content-based publish/subscribe middleware satisfies the real-time requirements of ESP applications (Stonebraker et al., 2005).

### Parallelization and speculative execution

Regarding the STM-based parallelization, we have two scenarios that illustrate its benefits in common cases. Both scenarios are based on the system of Figure 1. The first scenarios consider a case that there is no available parallelism in the workload (and/or in the semantics of the component), but there can still be some gain in latency. In this case we assume a system in which messages can be delivered out of the order of interest for the stateful component. Further, we assume the usage of a failure-awareness mechanism that can signal when a certain subset of the messages is guaranteed to be final (or a time-out that allows late messages to be discarded).

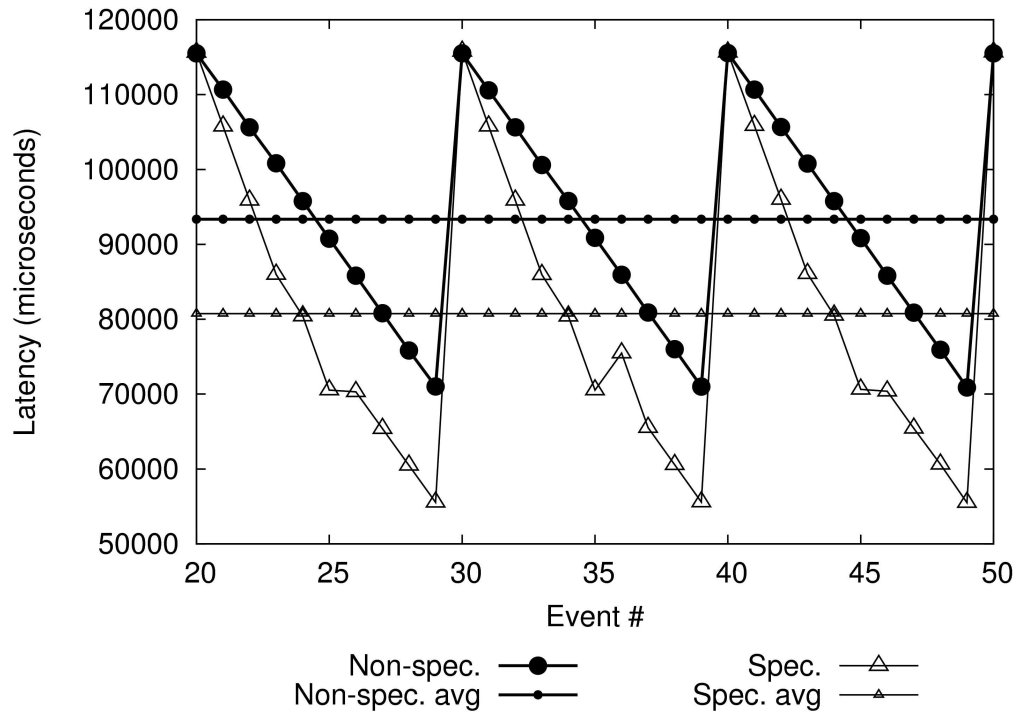


Figure 14. End-to-end latency comparison.

The end-to-end latency of individual events is shown in Figure 14. The behavior of the system depicted in the figure is periodic. In this example, events 1 to 9 (and then 11 to 19 and so on) are received but must wait until some information piggy-backed on event 10 (respectively, 20, 30 and so on) in order to be confirmed. As a consequence, the events modulo 10 are the ones that have higher latency as they need to wait a whole period

before committing. Similarly, events with timestamp  $ts \% 10 = 9$ , have the lowest end-to-end latency. Because the speculative system does some processing in advance, events (except for the module 10 events) have a lower end-to-end latency. An example of a wrong ordering and the resulting extra cost of a re-execution can be seen for event 36. For this experiment, the non-speculative system has an average latency 16% greater than the speculative system, as depicted by the two average lines in the figure.

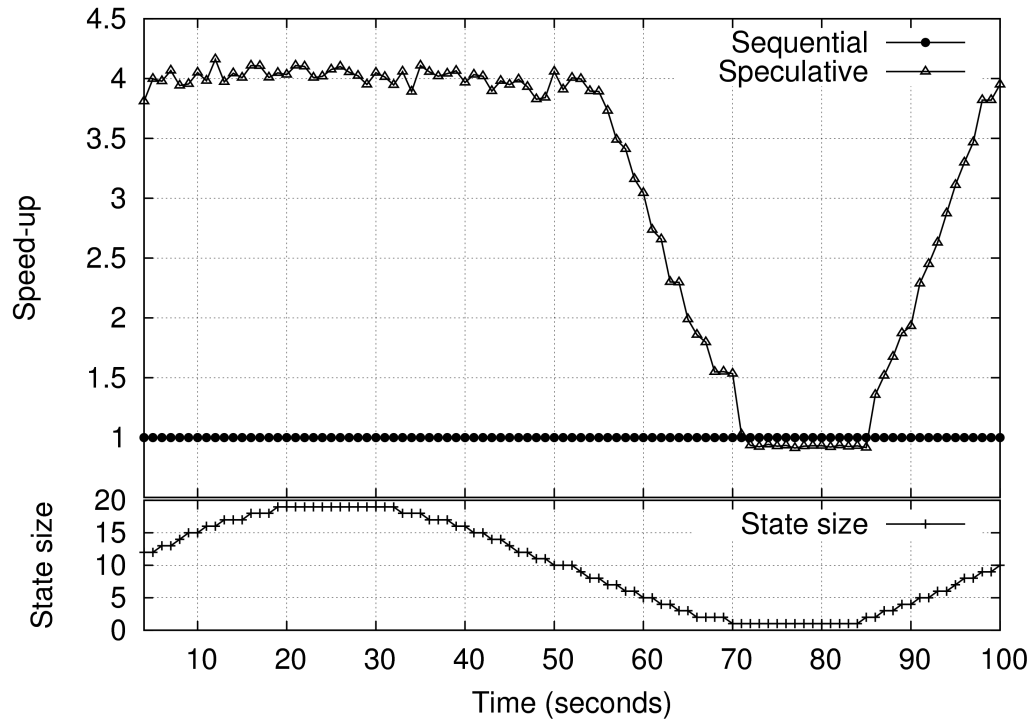


Figure 15. Throughput comparison for different levels of parallelism.

For the second scenario we consider that there is some parallelism available in the component and workload. More precisely, we consider that each event modifies one field of the components state. During runtime, we vary the number of unrelated fields in the component state, as depicted in the lower part of Figure 15: if the state contains only one field (the minimum value in the figure), any two events that execute concurrently will interfere because they modify the same field; on the other hand, if there are 20 fields (the maximum value), the probability that some events can execute in parallel without interfering is high. The speed up of the speculative system in comparison to the non-speculative one can be seen in the upper part of Figure 15. As expected, when the state size is big enough the speculation is able to increase the throughput and when the state size is one, the speculative version performs similar to the non-speculative version.

## CONCLUSION

StreamMine has several novel features. First, it uses speculation to parallelize the processing events. This permits us to spread the execution of CPU intensive components across multiple cores. Second, StreamMine uses a content-based pub/sub service to distribute the events among components. Our experience indicates that conventional event stream applications that are based on the pipe and filter paradigm are not easy to scale up. Using pub/sub, we can connect components via subscriptions instead of pipes. This increases the composability and scalability of StreamMine. Third, we extended the pub/sub mechanism to support load balancing across multiple machines.

StreamMine is under active development and we are currently implementing several applications on top of StreamMine to evaluate and evolve the APIs, the protocols, and our design decisions.

## REFERENCES

- Abadi, D. J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniak, M., Hwang, J.-H., Lindner, W., Maskey, A. S., Rasin, A., Ryvkina, E., Tatbul, N., Ying, & Y., Zdonik, S. The design of the borealis stream processing engine. In M. Stonebraker, G. Weikum. & D. DeWitt (Eds.), *Proceedings of the 2005 Conference on Innovative Data Systems Research* (pp. 277-289). Asilomar, CA.
- Aekaterinidis, I., & Triantafillou, P. (2007). Publish-Subscribe Information Delivery with Substring Predicates. *IEEE Internet Computing* 11(4), 16-23.
- Aguilera, M., K., Strom, R., E., Sturman, D., C., Astley, M., Chandra, T., D. (1999). Matching Events in a Content-Based Subscription System. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing* (pp. 53-61). Atlanta, Georgia, United States: ACM Press.
- Babcock, B., Babu, S., Datar, M., Motwani, R., & Widom, J. (2002). Models and issues in data stream systems. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (pp. 1-16). New York, NY, USA: ACM Press.
- Bloom, B. (1970). Space/Time Trade-offs in Hash Coding with Allowable Errors. In *Communications of the ACM*, 13(7), 422-426.
- Brito, A., Fetzer, C., Sturzhelm, H., & Felber, P. (2008). Speculative out-of-order event processing with software transaction memory. In R. Baldoni (Ed.), *Proceedings of the Second International Conference on Distributed Event-Based Systems* (pp. 265-275). New York, NY, USA: ACM Press.
- Cao, F., & Singh, J.P. (2005). MEDYM: Match-Early with Dynamic Multicast for Content-Based Publish-Subscribe Networks. In *Middleware 2005* (pp. 292-313). Berlin/Heidelberg, Germany: Springer Verlag.
- Carriero, N., & Gelernter, D. (1989). Linda in context. *Communications of the ACM*, 32(4), 444-458.

- Campanile, F., Coppolino, L., Giordano, S., & Romano, L. (2008). A business process monitor for a mobile phone recharging system. *The EUROMICRO Journal of Systems Architecture*, 54(9), 843-848.
- Carzaniga, A., Rosenblum, D., Wolf, A. (2001). Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3), 332-383.
- Carzaniga, A., & Wolf, A. (2003). Forwarding in a Content-Based Network. In *Proceedings of ACM SIGCOMM 2003* (pp. 163-174). New York, NY, USA: ACM Press.
- Cherniack, M., Balakrishnan, H., Balazinska, M., Carney, D., Çetintemel, U., Xing, Y., & Zsonik, Y. (2003). Scalable distributed stream processing. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*. Asilomar, CA.
- Cutting, D., Quigley, A., & Landfeldt, B., (2008). SPICE: Scalable P2P implicit group messaging. *Computer Communications*, 31(3), 437-451.
- Dean, J. (2006). "Experiences with MapReduce, an Abstraction for Large-Scale Computation", Keynote PACT06.
- Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1), 107-113.
- Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A-M. (2003). The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2), 114-131.
- Fabret, F., Jacobsen, H.A., Llirbat, F., Pereira, J., Ross, K.A., Shasha, D. (2001). Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data* (pp. 115-126). New York, NY, USA: ACM Press.
- Fan, L., Cao, P., Almeida, J., Broder, A.Z (2000). Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3), 281-293.
- Fidler, E., Jacobsen, H.A., Li, G., Mankovski, S. (2005). The PADRES Distributed Publish/Subscribe System. In *Feature Interactions in Telecommunications and Software Systems* (pp. 12-30).
- Herlihy, M., Eliot, J., & Moss, B. (1993). Transactional memory: Architectural support for lock-free data structures. In *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on* (pp. 289-300). New York, NY, USA: ACM Press.
- Hwang, J.-H., Çetintemel, U., & Zdonik, S. Fast and Highly-Available Stream Processing over Wide Area Networks. In *Proceedings of the 24th International Conference on Data Engineering* (804-813): IEEE Computer Society.
- Jerzak, Z., & Fetzer, C. (2007). Prefix Forwarding for Publish/Subscribe. In *DEBS '07: Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems* (pp. 238-249). Toronto, Canada: ACM Press.
- Jerzak, Z., & Fetzer, C. (2008). Bloom Filter Based Routing for Content-Based Publish/Subscribe. In *DEBS '08: Proceedings of the second international conference on Distributed event-based systems* (pp. 71-81). Rome, Italy: ACM Press.

- Koparanova, M., & Risch, T. (2004). High-performance grid stream database manager for scientific data. In F. Fernández Rivera, Marian Bubak, A. Gómez Tato & Ramon Doallo (Eds.), *European Across Grids Conference* (pp. 86-92). Berlin/Heidelberg, Germany: Springer Verlag.
- Mühl, G., Fiege, L., Pietzuch, P., (2006). *Distributed Event-Based Systems*. New York, NY, USA: Springer-Verlag.
- Oki, B., Pfluegl, M., Siegel, A., Skeen, D. (1993). The information bus -- an architecture for extensible distributed systems. In *Proceedings of the 14th Symposium on the Operating Systems Principles* (pp. 58-68). New York, NY, USA: ACM Press.
- Pereira, J., Fabret, F., Llibat, F., Shasha, D. (2000). Efficient Matching for Web-Based Publish/Subscribe Systems. In *CoopIS '02: Proceedings of the 7th International Conference on Cooperative Information Systems* (pp. 162-173). London, UK: Springer-Verlag.
- Pietzuch, P. (2004). Hermes: A Scalable Event-Based Middleware. *Computer Laboratory, Queens' College, University of Cambridge*, PhD Thesis.
- Rosenblum, D., & Wolf, A. (1997). A design framework for Internet-scale event observation and notification. *SIGSOFT Software Engineering Notes*, 22(6), 344-360.
- Shavit, N. & Touitou, D. (1995). Software Transactional Memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, 204-213.
- Spring, J. H., Privat, J., Guerraoui, R., & Vitek, J. (2007). Streamflex: high-throughput stream programming in java. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications* (pp. 211-228). New York, NY, USA: ACM Press.
- Sterling, T., Becker, D. J. and Savarese, D., Dorband, J. E., Ranawake, U. A., & Packer, C. V. (1995). BEOWULF: A parallel workstation for scientific computation. In D. P. Agrawal (Ed.), *Proceedings of the 24th International Conference on Parallel Processing*. London: CRC Press.
- Stonebraker, M., Cetintemel, U., Zdonik, S, (2005). The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4), 42-47.
- Tarkoma, S. (2008). Dynamic filter merging and mergeability detection for publish/subscribe. *Pervasive and Mobile Computing*, 4(5), 681-696.
- Tarkoma, S., Kangasharju, J. (2006). On the cost and safety of handoffs in content-based routing systems. *Computer Networks*, 51(6), 1459-1482.
- Tryfonopoulos, C., Zimmer, C., Weikum, G., Koubarakis, M. (2007). Architectural Alternatives for Information Filtering in Structured Overlays. *IEEE Internet Computing*, 11(4), 24-34.
- Welsh, M., Culler, D. E., & Brewer, E. A. (2001). Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)* , (pp. 230-243). New York, NY, USA: ACM Press.
- Zhao, Y., Sturman, D., Bhola, S. (2004). Subscription Propagation in Highly-Available Publish/Subscribe Middleware. *Lecture Notes in Computer Science*, 3231, 274-293.

## **KEY TERMS & DEFINITIONS**

**Bloom filter:** a probabilistic data structure able to test whether an element is contained within a set or not. A single Bloom filter is able to store a whole universe of elements.

**Conflict predictor:** is a module that controls the amount of speculation in a processing component that uses optimistic parallelization.

**Publish/Subscribe:** a communication paradigm where the communicating parties are decoupled in terms of time, space and synchronization. Obsoletes the need for source and destination addresses on data messages.

**Stateful component:** is a component that uses not only the current input, but also a state derived from previous computations, in order to execute its computations.

**STM (Software Transaction Memory):** is a programming construct that allows blocks of code to be executed in a way that appears to be atomic.

- i Author supported by the Polish Ministry of Science and Higher Education grant number N N516 375034.
- ii StreamMine is a part of larger EU project (FP7-216181) on Scalable Automatic Streaming Middleware for Real-Time Processing of Massive Data Flows (STREAM) - <http://www.streamproject.eu/>.
- iii For more details see: <http://wordlist.sourceforge.net>
- iv As defined in the `umontreal.iro.lecuyer.randvar.ParetoGen` package – see <http://www.iro.umontreal.ca/~simardr/ssj/> for more details.